TESIS DOCTORAL

Comprobación de modelos guiados por estrategias en lógica de reescritura
Model checking of strategy-controlled systems in rewriting logic

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Rubén Rafael Rubio Cuéllar

Directores

Narciso Martí Oliet
Isabel Pita Andreu
José Alberto Verdejo López

Madrid

Universidad Complutense de Madrid

Facultad de Informática

Tesis doctoral

Comprobación de modelos guiados por estrategias en lógica de reescritura

Model checking of strategy-controlled systems in rewriting logic

Memoria para optar al grado de Doctor

presentada por

Rubén Rafael Rubio Cuéllar

Directores

Narciso Martí Oliet
Isabel Pita Andreu
José Alberto Verdejo López

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Comprobación de modelos guiados por estrategias en lógica de reescritura

Model checking of strategy-controlled systems in rewriting logic

MEMORIA PARA OPTAR AL GRADO DE DOCTOR EN

INGENIERÍA INFORMÁTICA PRESENTADA POR

Rubén Rafael Rubio Cuéllar

DIRECTORES

Narciso Martí Oliet
Isabel Pita Andreu
José Alberto Verdejo López

Madrid, septiembre de 2021

Agradecimientos

He de admitir que he curioseado algunos agradecimientos de tesis para escribir estos, con los que doy fin a esta memoria (cronológicamente, lo siento, aún quedan 300 páginas). Intentaré huir de la autobiografía, del ensayo y del cantar de gesta. Sin embargo, no quiero zafarme de incluir nombres, aunque eso suponga omitir otros, porque le restaría valor al reconocimiento que se merecen. Entiéndase este párrafo como una disculpa para todos aquellos que no son mencionados.

En primer lugar, quiero agradecer a mis directores de tesis, Narciso, Isabel y Alberto, todo su trabajo y apoyo durante estos años. No solo han leído numerosas páginas con más erratas de las que quisiera, recibido largos correos y firmado abundantes informes, sino que me han enseñado a desenvolverme como investigador, cuando hubiera dicho que eso era cosa de Herlock Sholmes y Jessica Fletcher. Narciso me animó desde el principio a hacer el doctorado y se ha esforzado en que lo realizase en las mejores condiciones, a gusto y convencido.

A mis compañeros del despacho 219 y en especial a Adrián, Enrique y Manuel. Ellos han sido mis *ciceroni* en la universidad, alegrando el ambiente a base de paseos periunédicos y juegos de palabras, y me han dejado involucrarme en sus proyectos (Juan inclusive). Gracias a Óscar y Luis, pues aunque no hayamos trabajado juntos, nuestros temas se han retroalimentado. No me puedo olvidar tampoco de Pablo ni de Gorka.

To the members of the Maude Team, Paco Durán, Santiago Escobar, Steven Eker, Narciso again, José Meseguer, and Carolyn Talcott, who have welcomed me warmly and generously into their group. During this time, we have published two versions of Maude and a centenary article, committed to writing two books together, and met in different conferences and numerous Skype hours. I also appreciate Steven's confidence in letting me contribute to the well-kept Maude code, although the circumstances did not allow us to collaborate in person.

A Javier Esparza por aceptar ser mi anfitrión en medio de una pandemia y conseguir que haya disfrutado no ya de una estancia normal sino magnífica. I have learned and enjoyed a lot in his group, and have been reminded that thinking is not neccesarily an individual activity and how challenging and pleasant finding out a proof could be. Thanks to all the physical and virtual inhabitants of the seventh chair for making me feel like one of them: Bala, Roland, Chana, Martin, Philipp, Debarghya, Christoph, Stefi, Maxi, Mikhail, Muqsit...

He tenido profesores estupendos desde que me alcanza la memoria, pero como he prometido centrar el tema mencionaré únicamente a los que pueda relacionar estrechamente con la tesis. Fue con Ricardo con quien descubrí la especificación algebraica y oí por primera vez la palabra Maude (no seguida de Flanders). Su entusiasmo por la asignatura y los lenguajes de programación fue contagioso. Mis andanzas al otro lado del aula también empezaron como ayudante suyo y me he quedado con su sabio consejo de que la misión del profesor es abrir la cabeza a los estudiantes (en el buen sentido, claro está). Con David de Frutos continué con el tema leyendo el libro de Ehrig y Mahr y utilicé Maude por primera vez. Agradezco su apoyo en numerosas ocasiones, sus explicaciones sobre los protocolos poblaciones y sus ofertas para colaborar científicamente, que lo urgente no me ha permitido atender como debiera.

A Salvador Lucas, por los interesantes y fructíferos intercambios de correos sobre estrategias de normalización sensibles al contexto, que han dado lugar a la sección 7.1.

Como dice el himno, *vivat Academia*, *vivant membra quaelibet*. En particular, vivan los del departamento de Sistemas Informáticos y Computación, a los que agradezco haberme facilitado compaginar clases, estancias y cambios contractuales. Entre ellos Paco, que siempre me ha animado a quedarme en la universidad, y los profesores con los que he tenido el placer de coincidir en las clases prácticas. Gracias a mis coétaneos compañeros doctorandos Cristina,

Dani, Alicia, Miguel, Jesús, Pablo, Luisma, Marta, Toni, etc, por hacer este trance mucho más agradable.

Agradezco su disposición a participar en esta tesis y sus comentarios a los miembros del tribunal y a los evaluadores externos, Dorel Lucanu y Kyungmin Bae. Mención merecen también los revisores anónimos que han leído nuestros artículos y de los que siempre hemos obtenido comentarios provechosos.

Por último y no menos importante, a mis padres.

Muchas gracias.

La investigación descrita en esta memoria ha sido financiada por el Ministerio de Ciencia e Innovación y sus predecesores a través de los proyectos TRACES (TIN2015-67522-C3-3-R) y ProCode (PID2019-108528RB-C22), por la Comunidad de Madrid mediante el programa N-Greens Software-CM (S2013/ICE-2731), y por el Ministerio de Universidades y sus predecesores a través del contrato predoctoral FPU17/02319 y la ayuda para estancias breves EST21/00536.

....

This work has been funded by the Spanish Ministry of Science and Innovation and its predecessors through the projects TRACES (TIN2015-67522-C3-3-R) and ProCode (PID2019-108528RB-C22), by the Region of Madrid via the program N-Greens-Software-CM (S2013/ICE-2731), and by the Spanish Ministry of Universities and its predecessors through the predoctoral contract FPU17/02319 and the travel grant EST21/00536.

Abstract

Formal methods in computer science are mathematically rigorous techniques to develop and verify the correct behavior of hardware and software systems with different degrees of automation. Usually restricted to the analysis of critical systems, their application has become popular with the advance of techniques and has been extended to other disciplines as well. Computer-aided formal verification requires precise and expressive languages to describe the behavior of the system and the properties in question, along with algorithms to check or help to check whether these are satisfied.

Rewriting logic, proposed by Meseguer in 1992 as a unified model for describing concurrent systems, is an example of this. The states of the system to be modeled are described by terms in an equational logic and their transformations are represented by rules that rewrite those terms. Specifications in rewriting logic are not mere abstract models, but are executable and analyzable in specification languages such as Maude. An execution consists of a succession of independent rewriting steps, in which any rule is applied in any position of the term obtained in the previous step. This spatial and temporal locality is what confers the nondeterministic and concurrent character of the model. However, sometimes it is convenient to consider only some of the possible executions of the system by limiting the application of the rules, either for efficiency or to express restrictions with their own semantic value. The fundamental resource to express those restrictions without altering the system of rules and states are rewriting strategies.

Strategies are a primitive and pervasive concept in computer science. In rewriting logic, these have been traditionally expressed in the logic itself, due to its reflective nature. Nevertheless, to facilitate their use, a strategy language was proposed for Maude in line with other strategy languages of the moment such as ELAN and Stratego. Included as another level of specification in Maude, it allows the logic of the system to be separated from its control. In the first part of this thesis, the implementation of this language was completed, multiple extensions were studied and applied to the formal specification of various examples related to programming languages, communication protocols, games, computational and biological models, etc.

Model checking is a verification technique based on the exhaustive analysis of model executions, essentially represented as a system of states and transitions labeled with various atomic properties. On the basis of these, the properties of its behavior that are to be checked are expressed, usually by means of temporal logics. In this context it is natural to consider strategies that restrict the executions of the model and to consider how this affects the satisfaction of the desired properties. In this thesis we have studied the problem of model checking for systems controlled by strategies and in particular those expressed in the Maude strategy language. The Maude interpreter includes a model checker for rewriting systems and properties expressed in Linear-time Temporal Logic, which we have extended to take the restrictions of the strategy into account. We have also developed connections with external checkers to verify properties in other temporal logics like CTL* and the μ-calculus, and to calculate quantitative properties of probabilistic systems. Finally, all this has been used to check relevant properties in various examples specified with strategies.

Resumen

En la ingeniería informática, los métodos formales son técnicas matemáticamente rigurosas y en parte automatizables para desarrollar y verificar el correcto funcionamiento de sistemas hardware y software. Habitualmente restringidos al análisis de sistemas críticos, su aplicación se ha popularizado con el avance de las técnicas y se ha extendido también a otras disciplinas. Una verificación formal automatizada necesita lenguajes precisos y expresivos para describir el funcionamiento del sistema y sus propiedades en cuestión, junto con algoritmos que comprueben o ayuden a comprobar si estas se satisfacen.

La lógica de reescritura, propuesta por Meseguer en 1992 como un marco unificado para especificar sistemas concurrentes, es un ejemplo de ello. Los estados del sistema se modelan como términos en una lógica ecuacional y los cambios de estado se describen mediante reglas que reescriben esos términos. Las especificaciones en lógica de reescritura no son meros modelos abstractos, sino que son ejecutables y analizables en lenguajes de especificación como Maude. Una ejecución es una sucesión de pasos de reescritura independientes, en los que una regla cualquiera se aplica en una posición cualquiera del término obtenido en el paso anterior. Esa localidad espacial y temporal es lo que confiere el carácter no determinista y concurrente del modelo. Sin embargo, en ocasiones es conveniente considerar solo algunas de las posibles ejecuciones del sistema limitando la aplicación de las reglas, bien sea por eficiencia o para expresar restricciones con valor semántico propio. El recurso fundamental para expresar esas restricciones sin alterar el sistema de reglas y estados son las estrategias de reescritura.

Las estrategias son un concepto omnipresente en la informática desde sus orígenes. En la lógica de reescritura, estas se han expresado tradicionalmente en el propio lenguaje debido a su caracter reflexivo. Sin embargo, para facilitar su uso se propuso un lenguaje de estrategias propio para Maude en la línea de otros lenguajes del momento como ELAN y Stratego. Incluido como un nivel superior de especificación en Maude, permite desagregar la lógica del sistema de su control. En la primera parte de esta tesis se ha finalizado la implementación de este lenguaje, se han estudiado múltiples extensiones del mismo y se ha aplicado a la especificación formal de diversos ejemplos relacionados con lenguajes de programación, protocolos de comunicación, juegos, modelos computacionales y biológicos, etc.

La comprobación de modelos es una técnica de verificación basada en el análisis exhaustivo de las ejecuciones del modelo, esencialmente representado como un máquina de estados y transiciones etiquetados con diversas proposiciones atómicas. En base a ellas se expresan las propiedades de su comportamiento que se quieren comprobar, habitualmente mediante lógicas temporales. En este contexto es natural considerar estrategias que restrinjan las ejecuciones del modelo y plantearse cómo esto afecta a la satisfacción de las propiedades deseadas. En esta tesis hemos estudiado el problema de la comprobación de modelos para sistemas controlados por estrategias y en particular para aquellos controlados por el lenguaje de estrategias de Maude. Su intérprete dispone de un comprobador de modelos integrado para sistemas de reescritura y propiedades en lógica temporal lineal, que hemos extendido para tener en cuenta las restricciones de las estrategias. Además hemos desarrollado conexiones con comprobadores externos para verificar propiedades en otras lógicas temporales como CTL* y el μ-cálculo y calcular propiedades cuantitativas en sistemas probabilísticos. Finalmente, todo ello ha sido utilizado para comprobar propiedades interesantes en diversos ejemplos especificados con estrategias.

Contents

List of Figures

List of Tables

List of Notation

Σ^*	Finite words over an alphabet Σ	2.1
Σ^ω	Infinite words over an alphabet Σ	2.1
Σ^∞	Finite and infinite words over an alphabet Σ	2.1
$\inf(\pi)$	States repeated infinitely often in the execution/run π	2.1
$\mathcal{P}(X)$	Parts of the set X	2.5
$\overline{L}$	Language or topological closure of L	2.5
$\Gamma^*_{\mathcal{A},s}$	Finite executions of $\mathcal{A}$ from the state s (if omitted, from any state)	2.2
$\Gamma^\omega_{\mathcal{A},s}$	Infinite executions of $\mathcal{A}$ from the state s	2.2
$\Gamma_{\mathcal{A},s}$	Executions of $\mathcal{A}$ from the state s	2.2
$\mathcal{A}_H$	Stuttering extension of $\mathcal{A}$ on the halting states H	2.2
λ	Usually, an intensional strategy	2.2
E	Usually, an extensional strategy or a set of equations	2.2
$E(\lambda)$	Extensional strategy denoted by λ	2.2
λ_E	Intensional strategy that denotes $\overline{E}$	2.2
$L(\varphi)$	Language of propositional traces over $\mathcal{P}(AP)$ denoted by the linear φ	2.3
π^k	Suffix of the word/execution/run π from its k-th position	2.3.1
$T_\Sigma(X)$	Terms over the signature Σ with variables in X	2.4
T_Σ	Ground terms (without variables) over the signature Σ	2.4
$T_{\Sigma/E}$	Initial algebra of the equational theory (Σ, E)	2.4
$t[x/u]$	Substitution of every occurrence of x in t by u	2.4
$t\vert_p$	Subterm of t at position p	2.4
$t[p/u]$	Replacement of the subterm at position p in t by u	2.4
$\to^1_R$	One-step rule rewrite	2.4
$\rightsquigarrow$	Partial function	2.6
$\mathcal{P}_\perp(X)$	Power domain for the results of the denotational semantics	3.4
$[\![\alpha]\!]_\Delta$	Denotation of the strategy expression α using the definitions Δ	3.4
$g \circ f$	Functional composition of g and f with $(g \circ f)(x) = g(f(x))$	2.4

Part I

Introduction

Chapter 1

Introduction

Computer systems are not exempt of errors, as the humans who design and program them are liable to make mistakes. Reliability is one of the main concerns of engineering, although the required confidence level may be adjusted in proportion to the severity of a failure. Most programs we routinely use in our desktop or mobile computers are updated continuously to correct bugs (and probably introduce new ones) that are usually no more than annoying for those who have suffered and perhaps reported them. Their mild repercussions allow asking users for their collaboration as guinea pigs, making the number of users an informal measure of reliance. However, hardware cannot be updated as easily and critical software needs more attention when its defects can lead to serious economic consequences or even affect human life and integrity. For sure, crashing hundreds of airplanes cannot be part of the development of their navigation system, even if they are empty. Expending time and effort to test them systematically and construct mathematical proofs of their properties is then worthwhile. In this context, *formal methods* are a wide range of mathematically rigorous techniques to describe the behavior and expected properties of hardware and software systems and to verify them. Quoting [McM93],

> *Formal verification means (1) having a mathematical model of a system, (2) a language for specifying desired properties of the system in a concise, comprehensible and unambiguous way, and (3) a method of proof to verify that the specified properties are satisfied. When the method of proof is carried out substantially by machine, we speak of automated verification.*

The main topic of this thesis is a particular instance of formal verification where the three components are respectively rewriting logic controlled by strategies, temporal logic, and model checking. All of these topics are integrated into the Maude specification language, and they will be visited in the following lines.

Formal specification. Awareness of “the correctness of the results, united with the economy of time” when doing mathematical calculations was behind the development of a variety of mechanical calculating devices throughout history. The first published algorithm designed to be executed by a machine is attributed to Ada Byron for her notes on the hypothetical Charles Babbage’s Analytical Engine in 1842 [Men42]. In these notes, an algorithm to compute Bernoulli numbers is dissected and justified step by step. A century later, that was also a concern for the pioneers of modern computing Alan Turing and John von Neumann, who proposed tentative methods to prove the correctness of programs [MJ84]. However, the development of the basic instruments for describing and reasoning about programs began around the 1970s with the modern abstract data types and the formal semantics of programming languages [EM85]. The role of semantics is assigning precise meaning to programs, and this is the title of a work by Robert Floyd [Flo67] where he observed that imperative programs can be interpreted as flowcharts and annotated with predicates describing the relations between the values of their variables at each step. These predicates are transformed by the programming language statements according to some axioms proposed by Tony Hoare [Hoa69] in the logic that takes his name, while claiming that "computer programming is an exact science."

Programs can also be given meaning as mathematical functions like in denotational semantics [SS71], or by specifying how their executions change step by step the state of a machine as in operational semantics, first used for describing the programming language Algol68 [Plo04]. Furthermore, complex programs require using complex structures to organize and process their data: lists, queues, trees, sets, tables, etc. Concentrating on their abstract values and operations, they are modeled as *abstract data types* [LZ74]. Since the science of combining elements and operations under certain principles is algebra, these structures can be given convenient *algebraic specifications* [EM85]. For example, a list may be described either as an empty list *nil* or as its first element x followed by the rest xs of the list $cons(x, xs)$. Then, a property of the list like its length can be defined with equations on these symbols $length(nil) = 0$ and $length(cons(x, xs)) = 1 + length(xs)$. Actually, the frontier between program and data is not rigid, and an algebraic specification can be understood as a program whose states are the terms rewritten by the equations, if they are well-behaved enough. Unifying program and data leads to rewriting systems [BN98, Ter03] and functional programming languages. The mentioned approaches are less practical when dealing with *concurrent systems* composed of multiple parts that can be executed out-of-order interacting with each other. This is the case of computer networks sharing common resources, multiprocessing operating systems, multithreaded programs, web services, databases, etc. Concurrency produces a huge number of possible execution paths without clearly predictable interleavings, for which specific methods are advisable. A variety of modeling languages and process algebras have been proposed like Petri nets [Pet62], Hoare's CSP [Hoa85], Milner's CCS [Mil80], and the actor model [HBS73]. However, these formalisms do not provide a unified approach to all concurrency problems, for which unifying theories like chemical abstract machines [BB92], π-calculus [MPW92], and rewriting logic appeared.

Rewriting logic. Rewriting logic [Mes92] was proposed by José Meseguer in 1992 as a unified model of concurrency based on equational logic and term rewriting. Algebraic terms related by equations describe the state of the concurrent system, and change of state is represented by rules that rewrite these terms. A rewriting system is executed by the successive application of rewrite rules in arbitrary positions of the term modulo the equations, both chosen nondeterministically. Concurrency is captured naturally, since multiple rules can be enabled at the same time in possibly different fragments of the term. Moreover, the logic has been proven useful as a framework to describe other logics and semantics [MM96, MPV07]. Rewriting logic specifications are executable, and indeed, the logic was introduced together with a language called Maude to execute and analyze them. Maude [CDE+20, CDE+07] is a multiparadigm specification and programming language that supports modular functional specification as in the OBJ3 language [GWM+00] introduced by Joseph Goguen. These *functional modules* can be extended with rules whose application means a nondeterministic transformation of the state, instead of an identity between two terms used to represent it. The result is called a *system module* and represents a full rewrite theory. Rewrite rules are not necessarily confluent or terminating unlike equations, that is, they may not lead to the same result however they are applied, and its application may not necessarily finish. There are other languages close to rewriting logic like CafeOBJ [DF02] and ELAN [BKK+01].

Rewriting logic and Maude have been the subject of extensive research and applications during its almost thirty years [Mes12, MPV12]. One of their most used features is that they are reflective languages where their programs, operations, and other aspects of its metatheory can be represented. Terms, equations, rules, modules, and also matching, equational reduction, rule application, and so on are represented in a *universal theory* collected in the `META-LEVEL` module of the Maude prelude. This possibility has been fruitfully used in Full Maude [CDE+20; Part II], an extensible Maude interpreter written in Maude itself with additional constructs and commands, and to analyze the Maude specifications themselves with tools like the Maude Formal Environment [DRÁ11]. Recent trends in Maude are related to symbolic computation [DEE+20] by means of unification and narrowing, which are applied for example to the Maude-NPA [EMM09] tool for cryptographic protocol verification, and also with the integration of external SMT solvers.

Strategies. The spatial and temporal locality of rules is the cornerstone of the natural and simple representation of concurrency in rewriting logic. However, it is sometimes convenient to tame this nondeterminism to capture additional restrictions, the global behavior of the system, or to direct rewriting efficiently to a desired goal. For example, the terms and deduction rules of an inference system can be described as a rewrite theory and be proven sound, but only a careful application of these rules leads to the desired deductions. Expressing the additional restrictions at a higher level on top of the base specification facilitates understanding, reusability, and compositionality, according to the principle of separation of concerns [Dij82]. This idea is enunciated in Kowalski's motto *Algorithm* = *Logic* + *Control* [Kow79] and developed in Lescanne's *Rule* + *Control* approach [Les90], which promote the separation of the concerns of rules and their control. Strategies are the representation of this control, and they are pervasive in computer science, from the λ-calculus [Bar14] to artificial intelligence and game theory. In term rewriting and functional programming, strategies are usually implicitly given as statements like "reduce the innermost rightmost position first" or "evaluate the arguments of a function before the function itself" that receive meaningful names like *applicative strategy* or *call-by-value* [BN98, Ter03, BCD+09]. However, programmable strategies allow describing arbitrary control mechanisms explicitly using appropriate strategy languages. These strategies are a central part of the rewriting-based specification language ELAN [BKK+01], which starts using them for formal specification. Other strategy languages appeared at that moment like Stratego [BKV+08] for rule-based program transformation, and Tom [BBK+07] that pushes algebraic types and rewriting to Java. Some others have originated more recently like ρLog [MK06], and Porgy [FKP19] for graph instead of term rewriting.

In Maude and rewriting logic, strategies have also been present to control rewriting from their very beginning [CM96, CM97]. Thanks to reflection, strategies can be expressed with a great flexibility inside Maude, whose universal theory includes symbols that represent the application of a rule to a term and other associated operations. For example, the strategy informally expressed as "try to apply r_1, and if it fails, try again with r_2" can be described as the following Maude term:

```
eq applyR1R2(M, T, N) =
  if metaXApply(M, T, 'r1, none, 0, unbounded, 0) =/= failure then
    metaXApply(M, T, 'r1, none, 0, unbounded, N)
  else
    metaXApply(M, T, 'r2, none, 0, unbounded, N)
  fi .
```

Although the syntax and semantics of Maude have not been explained yet, we may get an intuition of its meaning or at least notice that a simple statement has yielded a somehow verbose and not easily understandable expression. In order to evaluate the strategy, we would also need to operate with the reflective representation of the terms and Maude programs involved, which may be seen as a barrier for users that are not conversant with the Maude metalevel. Hence, different high-level strategy languages have been proposed and implemented in Maude itself using reflection [CEL+96, CM97]. After this experience and coinciding with the appearance of other strategy languages like ELAN and Stratego, the goal of providing a basic strategy language for Maude was undertaken [MMV04]. Its design was influenced by these strategy languages, but it promotes a clear separation between rules and strategies, which are instead tightly coupled in ELAN, so that the same rewriting system can be easily controlled with alternative strategies. Moreover, it does not ambition to be complete and include many combinators as Stratego, since reflection can always be used to extend it if required. For example, the previous property can be expressed in the language as `r1 ? idle : r2` with a conditional operator, or simply as `r1 or-else r2`. The language was first prototyped within Full Maude, which provides a user-level interface to specify and execute strategies, and it has been applied to many non-trivial examples regarding semantics of programming languages [HVO07, MPV07, RSV06], deduction procedures [VM11, EMM+07], and telecommunication networks [PM02], among others. Not all its applications are strictly related to specification, but strategies can also be used for programming for example a Sudoku solver [SP07] and neural networks [SPV09].

After testing the language in its prototype form, a more efficient C++ implementation as integral part of the Maude interpreter was started by Steven Eker [EMM+07], whose completion was the first objective of this thesis and which was incorporated to the official 3.0 release. While the implementation was already advanced, some features were missing including some important strategy combinators, and the ability to declare strategies compositionally in strategy modules and execute them recursively. Strategy modules constitute a third level of specification on top of functional and system modules, and they enjoy the same possibilities including parameterization [RMP+19c] and reflection. The strategy language is represented at the metalevel too, allowing flexible transformation and generation of strategies that are efficiently executed by the interpreter. Among other applications, this has been used to build extensions of the strategy language with additional combinators and a framework for concurrent and distributed strategies (see Chapter 7). The semantic foundations of the strategy language [EMM+07, MMV09] have also been extended with complete and updated definitions and a new equivalent small-step operational semantics that highlights the rewriting paths described by a strategy. This information will become relevant for model checking. All known examples in the previous versions of the strategy language have been tested and sometimes enhanced with the new implementation, and various other examples have been specified and programmed. Some of them are described in Part III.

Temporal properties. The first step of the formal verification agenda, a mathematical model of the system under study, has already been addressed. In order to formally assess whether the desired properties of the system are satisfied using it, we need to express them too in an equally precise form. Given a collection of basic predicates about the states of the model, propositional logic can be used to express more complex properties on the states, like invariants. However, state and transition models are essentially dynamic, and we want to describe their behavior as transitions take place. In the philosophical context, modal logics arose to express nuances in logical statements regarding necessity, possibility, knowledge, or time with additional operators called modalities [vBen10]. Temporal logics [Eme90] follow this principle and extend propositional logic with temporal operators like *eventually*, *always*, or at the *next* step to specify how facts change over time, which is usually considered a discrete measure. Widely used logics in formal verification are Linear-time Temporal Logic by Amir Pnueli [Pnu77] and Computational Tree Logic by E. M. Clarke and E. A. Emerson [CE81], which are combined in their superset CTL* [EH86]. A less intuitive but more expressive logic is μ-calculus [Koz83], whose modal operators allow referring also to the actions that change the state. Model checking, which is explained in the following paragraphs, can be used to check whether these properties hold.

Formal verification. Once the behavior and intended properties of a system are expressed in an appropriate language, decision procedures are required to check whether they meet. Ideally, this check would be completely carried out by a machine, hence saving a lot of tedious work, making the methods accessible to more people without specific training, and avoiding possible mistakes and distrust. However, this is very unlikely since any non-trivial semantic property of any program cannot be automatically decided, as proven by Rice's theorem. For that, we would need a machine like the *calculus ratiocinator* envisioned in the seventeenth century by Gottfried Leibniz that given any mathematical statement tells whether it is valid or not, in the same way other mechanical machines calculate arithmetical operations [Dav01]. The existence of such a device was posed as a challenge known as the *Entscheidungsproblem* (decision problem, in German) by David Hilbert and Wilhelm Ackermann in 1928, which Alan Turing and Alonzo Church answered negatively eight years later after formalizing the intuitive notion of what can be effectively calculated in Turing machines and the λ-calculus respectively. For their equivalent and universally accepted notions of algorithm, there are *undecidable* problems that cannot be solved by any algorithm. Hence, verification techniques must choose a compromise between automation, expressiveness, and completeness. At the lower end of automation, interactive theorem provers essentially accompany the users who specify the steps of the proof by themselves, by certifying the correctness of their deductions, revealing failures or requiring more details. Coq [Coq21], Isabelle/HOL [NPW02], and the Maude's ITP tool [CPR06] at a different scale are examples of proof assistants that have been applied to the verification of many computer and non-computer related systems. On the opposite side, SMT (satisfiability modulo theories) solvers, termination checkers, symbolic execution and other similar techniques can be used in a completely unattended way, assuming that they would not always finish with an answer unless they operate on a very restricted context. The advantage of these latter methods is that they run without dedicated effort and knowledge of the programmer, so they can be integrated into different programming tools outside academia. For instance, the Ada programming language has traditionally included various forms of static analysis, the smart-contracts language Solidity resorts to SMT solvers to prove conditions on its programs, and recently the mainstream compilers GCC and Clang incorporate static analysis modules for popular languages like C++. In those compilers, the user is not required even to give any specification of the desired properties, but only off-the-self ones can be checked. Moreover, their foundations are not usually formal but informal semantics of the languages. More semantically-accurate methods can be found in the $\mathbb{K}$ framework [RS10], which is partially based on Maude.

Model checking. Model checking [CHV+18] is a verification technique in the middle ground of automation, introduced by E. M. Clarke and E. A. Emerson [CE81], and by J. P. Queille and J. Sifakis [QS82], for which three of them received the Turing award in 2007. In the classical setting, a model described as a state and transition system is checked against a property expressed in a temporal logic, by the exhaustive exploration of its executions. These properties are constructed in terms of some atomic propositions that label the states or transitions of the model. Such an almost brute-force procedure can only conclude that an arbitrary property is satisfied when the model is finite. However, even if it is not, the procedure can be useful to reveal failures and provide counterexamples of the desired properties, although it is not guaranteed to terminate in this case. Model checking has had many industrial applications in the software, telecommunications, and specially in the hardware field, where finite systems are more common [HKK+13, Cra12, Wig08, NASA11, DKN+06, KNS02; ...]. The concepts behind model checking, the preparation of the model and the properties are relatively simple, while the verification algorithms do not require the user intervention. Hence, model checking is taught in many undergraduate courses of computer science, telecommunication, and other engineering studies. Although we have mainly focused on the classical setting, model checking is a wide term that encompasses diverse kinds of modeling frameworks and techniques. For example, time is a discrete measure in traditional models, but real-time systems can also be model checked with similar but specific procedures, and the same happens for probabilistic or stochastic systems. Many model checkers in active development are currently available with support for different temporal logics and modeling languages, like Spin [Hol+21], NuSMV [CCG+02], DiViNE [BBH+13] for software verification, mCRL2 [BGK+19] and TAPAS [CDL+08] based on process algebras, UPAAL [LYP+] for real-time systems, PRISM [KNP11] and VeStA [SV13a] for probabilistic and statistical model checking, and MCMAS [LQR17] for multi-agent systems.

However, decidability is not the only limitation to model checking and other formal verification methods. The complexity of the algorithmic problems involved, which rapidly grows due to their combinatorics, may make verification intractable or in a better case would require a significant amount of computing power and patience. Several alternatives have been proposed to reduce the state explosion problem, including abstraction to reduce the number of states by ignoring their irrelevant characteristics, partial order reduction to avoid exploring redundant executions [Pel93], and symbolic model checking [McM93]. Symbolic means here that the model states and transitions are represented symbolically as logical formulas, usually with the help of binary decision diagrams (BDD) or Boolean satisfiability (SAT) solvers. This is sometimes more efficient than the classical *explicit-state* model checking, where states are enumerated one by one.

The Maude LTL model checker [EMS04] is an explicit-state model checker for Linear-time Temporal Logic integrated into the Maude interpreter. Rewriting systems, as already mentioned, can be naturally seen as a transition system with terms in the role of states and one-step rules as transitions. Formulae are directly represented as terms in an appropriate signature, and atomic propositions are defined equationally on the states. The performance of the model checker is comparable to other standard tools, and it has been used to verify properties in multiple systems [MPV12, Rie18, FCM+04, CFP+18, Oga17, CM17; …]. In addition to the built-in one, other model checkers have been proposed for different logics and model formalisms within Maude. The Real-Time Maude [Ölv14] framework includes a model checker for Timed Computational Tree Logic (TCTL) [LÁÖ15], the linear-time sublogic of Meseguer's Temporal Logic of Rewriting can be checked under fairness conditions [BM15], as can μ-calculus properties [Wan04, Lec96]. True symbolic model checking where states may contain variables and are transformed by narrowing instead of rewriting can be achieved with the Maude logical model checker [BEM13], and statistical model checker by simulation can be applied to probabilistic rewrite theories specified in PMaude [AMS06].

Model checking strategy-controlled systems. Systems specified with strategies in Maude cannot be verified with its standard model checker, because it operates on the bare rewriting system and does not take strategies into account. Maude users would be reluctant to embrace the strategy language as a useful specification resource if such an interesting verification tool were no longer available. The main objective of this work is studying the model-checking problem for systems controlled by strategies, and implementing model checkers for those expressed using the Maude strategy language. The key principle of model checking in strategy-controlled systems is the straightforward observation that strategies designate a subset of executions of the model and that properties should be checked with only these executions in mind. For some simple notions of strategy, removing some transitions is enough to obtain an equivalent model satisfying the desired restrictions. However, our concept of strategy is highly expressive, since strategy expressions are arbitrary programs with rule application as its basic instruction. Effectively model checking these controlled systems would pass by transforming the model using convenient operational semantics to a standard structure where standard algorithms can be applied. The transformed system can be simpler, but also much more complex than the original model.

While the model-checking problem for strategy-controlled systems has not been explicitly studied in the literature, the basic ideas have been naturally used. In Upaal Stratego [DJL+15], properties can be checked in the "strategy space" of a stochastic priced timed game. Although through a different point of view, the combination of strategies and model checking is an active field of research in the context of multiplayer games or multi-agent systems where several very general strategy logics have been proposed [MMP+14, AHK02, CHP10, CBC15]. Their formulae include strategy quantification and their satisfaction depends on the existence of strategies for the player or actors such that the propositional part of the formula holds. Moreover, model checkers for reasonable simplifications of these logics do exist [LQR17, CLM+14], which are capable of synthesizing strategies as witnesses of the satisfaction of the formula.

The more practical outcome of the thesis is an extension of the Maude LTL model checker for rewriting systems controlled by the strategy language, which has been later extended using external model checkers to branching-time logics like CTL, CTL*, and μ-calculus. The implementation is based on a small-step operational semantics of the language so that the executions of the strategy-controlled model are guaranteed to be executions of the underlying system. Moreover, our model checker lets selected strategies be considered as atomic steps of the model, so that properties can be checked at different granularities and scales. Strategies are also a useful resource to capture parallel rewriting under different constraints, and parallel steps can be observed faithfully with this feature, as illustrated in Chapter 8 for the maximal parallel steps of membrane computing. In addition to the tandem strategy-based specification plus temporal property to be checked on it, the strategy-aware model checker can leverage strategies for the only sake of verification. Restrictions imposed by strategies can be used to reduce the state space, avoiding known redundant executions as a kind of partial order reduction, or directing the model checker search towards surmised counterexamples. Linear-time properties and other particular cases of branching-time properties are preserved by strategy constriction, so finding a counterexample in a controlled model reveals a counterexample of the original one. Moreover, such a counterexample will be tailored by the strategy, so it may be simpler and easier to understand.

Finally, the model checkers have been applied to multiple models of different complexity and nature, which has also served to evaluate the performance of the tools. Examples include deduction procedures, programming language semantics, games, alternative computational models, etc. In fact, the algebraic and logical methods used for the formal verification of hardware and software can be applied to other disciplines in a very similar way.

Probabilistic and quantitative verification. In addition to the qualitative and absolute statements usually expressed with temporal logics, quantitative properties like time and cost and less strict certainties involving probabilities are sometimes very useful. Model-checking techniques are available for this kind of verification, but to obtain sound numerical results from Maude models, their nondeterminism should be quantified. In the same way that strategies are helpful resources to limit nondeterminism, they are also appropriate to measure it [BÖ13]. On the one hand, we have extended the Maude strategy language with combinators that assign probabilities to the alternatives allowed by the strategy, which can then be simulated stochastically. On the other hand, we have extended the connection with external model checkers with the probabilistic tool PRISM [KNP11], so that we can calculate the probability that a PCTL [HJ94] or LTL formula holds, or the expected value of some Maude-defined function on the states. This is the most recent work included in this thesis, self-sufficient and already applied to examples like population protocols and chemical reaction networks, but not yet published and still in expansion.

1.1 Contributions of this work

The objectives and the main contributions of this work can be summarized in the following:

- The development of the Maude strategy language, whose implementation has been finished and incorporated to the official Maude 3.0 interpreter [CDE+20]. While the language already counted with a prototype written in Maude by Alberto Verdejo and a great part of its C++ implementation was already done by Steven Eker [EMM+07], we have added support for some of the most powerful and useful resources of the language like the `matchrew` operators for rewriting inside selected subterms, strategy modules with recursive strategies and parameterization, the reflection of the strategy language at the metalevel, the `dsrewrite` command for finding solutions of the strategy in a depth-first way, and the extension of the `search` command to restrict the search with a strategy and observe the intermediate states of strategy executions. Moreover, various extensions and extension patterns for the strategy language have been worked on. As a result, rewriting strategies can be easily expressed and efficiently executed in Maude and rewriting logic, without confusing the local meaning of rules with the overall control of the system provided by the strategies. We also compare the design decisions, expressivity, and syntax of the strategy languages ELAN, Stratego, Tom, and ρLog with Maude's.

- The advancement of the theoretical foundations of the strategy language, by defining a complete denotational semantics based on power domains to characterize the results of strategy computations. This is based on a simpler set-theoretic semantics introduced in the earlier papers of the language, which did not handle `matchrew` operators, recursive strategies, nontermination, and the finite failure of conditional expressions. In addition, we have defined a non-deterministic small-step operational semantics and proven it equivalent to the denotational one. This new semantics completely describes the rewriting paths allowed by a strategy, including their intermediate states and infinite executions. Seeing the set of allowed rewriting paths as a language, we have proven that the Maude strategy language is Turing complete and other related properties.

- The study of the model-checking problem for strategy-controlled systems. In an abstract setting, we first arrived to simple and natural definitions of the problem, which formalize the implicit ideas behind the semantics of strategic logics and related tools. The linear-time case is straightforward, and for the branching-time case we introduce the unwinding of a Kripke structure by a strategy as a reference for establishing the satisfaction of properties. In practice, we propose transforming the original module to encompass the restrictions of the strategy, and then apply standard algorithms on it. However, this procedure cannot be followed when strategies include a fairness component, so direct methods are discussed too. We also consider the associated satisfaction and synthesis problem, solving the linear-time case by the standard automata-theoretic approach and reducing the branching-time case to the model-checking problem of some strategy logics.

- The development of a model checker for strategy-controlled Maude specifications. We have applied the previous considerations to rewriting systems controlled by the Maude strategy language using the already mentioned small-step operational semantics, which identifies the rewrites of the underlying model as its atomic steps. An extension of the Maude LTL model checker has been developed to verify strategy-controlled specifications in the same conditions as the standard ones. In addition, the model checker allows considering selected strategies as atomic steps, so that systems can be easily checked at different granularities and parallel rewriting can be faithfully represented. As a digression, we provide a variation of the semantics so that strategies can express fairness constraints and models controlled by these strategies can be model checked too, for which a prototype implementation is available.

- The connection between Maude and external model checkers to verify CTL* and μ-calculus formulae on strategy-controlled and standard Maude specifications. Checking branching-time properties with strategies entails some theoretical difficulties that have been discussed. Moreover, the connection to Maude has evolved to a general-purpose library to operate with Maude entities in external programs. It has been used for other verification and visualization tools, and for integrating Maude into other programs and devices.

- The specification in Maude and its strategy language of multiple examples of systems from different areas, which have been used to test and benchmark the strategy language and its model checkers. Some have been adapted or extended from strategy-based specifications in Maude and other languages, and others have been written from scratch. These examples are related to deduction procedures, algorithms, semantics of programming languages, games, alternative computational models, etc.

- The development of a framework for extending Maude specifications with probabilities and having them verified by probabilistic model-checking methods, and an alternative probabilistic extension of the Maude strategy language intended mainly for stochastic simulations and statistical model checking. We think that these approaches are more intuitive and usable than PMaude. In the first case, verification is carried out by an external tool, connected to Maude using the infrastructure introduced for the previous external model checkers. This extension is recent work with respect to the other topics in this thesis, and there is more margin for extension and improvement. Nevertheless, we have applied them to the specification, verification, and simulation of population protocols and chemical reaction networks, among other examples.

1.2 Structure of this thesis

This document is structured in two main parts, *Strategies and model checking* and *Examples*, preceded by this introduction and state-of-the-art review, and followed by the conclusions and two appendices.

This section concludes the introduction, where the motivation and the main goals and contributions of the thesis have been highlighted. The following chapter looks deeper into and discusses the current development of the topics in which this work is based, including rewriting logic, strategies, and model checking. These concepts are explained at different levels of detail as they are required to understand the contributions of this thesis.

Chapter 3 is devoted to the Maude strategy language. After describing its syntax and informal semantics, two formal semantics of the language are provided. The first one is a more rigorous presentation of the typical description of the language, which focuses on the results of the strategy computations. The second one is a nondeterministic small-step operational semantics that emphasizes the intermediate execution states and their transitions, so that it can be the basis for the model checking of rewriting systems controlled with this language. The expressivity of the language and its implementation within the Maude interpreter are then discussed. Finally, the Maude strategy language is compared with other strategy languages in the literature.

Chapter 4 revolves around the model checking-problem for systems controlled by strategies. The discussion starts in an abstract setting with a general definition of the problem and some generic hints on how it can be addressed. These principles are applied to the Maude strategy language using the already introduced small-step operational semantics, and the extension of the Maude model checker for strategy-controlled systems is described. Branching-time properties are addressed afterward, and we present the `umaudemc` tool for evaluating them through external model checkers. Section 4.5 describes a prototype model checker for a variation of the strategy language semantics that allows expressing fairness restrictions in the strategy itself, Section 4.6 discusses the associated satisfiability problem, and related work is compared to close the chapter.

Chapter 5 introduces a probabilistic extension of the Maude strategy language and the model-checking tools in the previous chapter.

Part III describes various examples of rewriting systems controlled by the Maude strategy language. A variety of relatively small examples from different fields are covered in Chapter 6, some others related to extensions of the strategy language and reflection are gathered in Chapter 7, a strategy-based simulator and model checker for membrane systems is detailed in Chapter 8, and population protocols and chemical reaction networks are addressed in Chapter 9. Using these examples among others, the performance of strategy language implementation and the model checkers is evaluated in Chapter 10.

Finally, the conclusions are written in Chapter 11. The Maude language bindings that give support to the `umaudemc` tool are described in Appendix A, and Appendix B includes the proofs of all the propositions appearing in this thesis.

This dissertation reproduces what has been reported in various journal, conference, and workshop papers. Chapter 3 on the strategy language is based on

[DEE+20] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. Programming and symbolic computation in Maude. *J. Log. Algebraic Methods Program.*, 110, 2020. 58 pages.

[EMM+20] Steven Eker, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Alberto Verdejo. The Maude strategy language. *Manuscript*, 2020.

[RMP+19a] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Model checking strategy-controlled rewriting systems. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, 34:1–34:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[RMP+19c] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Parameterized strategies specification in Maude. In José Fiadeiro and Ionuț Țuțu, editors, *Recent Trends in Algebraic Development Techniques. 24th IFIP WG 1.3 International Workshop, WADT 2018, Egham, UK, July 2–5, 2018, Revised Selected Papers*, volume 11563 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2019.

Chapter 4 on strategy-controlled model checking derives from [RMP+19a] and

[RMP+20c] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Strategies, model checking and branching-time properties in Maude. In Santiago Escobar and Narciso Martí-Oliet, editors, *Rewriting Logic and Its Applications - 13th International Workshop, WRLA 2020, Virtual Event, October 20-22, 2020, Revised Selected Papers*, volume 12328 of *Lecture Notes in Computer Science*, pages 156–175. Springer, 2020.

[RMP+21a] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Model checking strategy-controlled systems in rewriting logic. *Automat. Softw. Eng.*, 2021. Accepted. 55 pages.

[RMP+21b] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Strategies, model checking and branching-time properties in Maude. *J. Log. Algebr. Methods Program.*, 123, 2021. 28 pages.

Some examples in Part III appear in [EMM+20, RMP+21a, RMP+19c, DEE+20] and

[CDE+20] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. *Maude Manual v3.1*. October 2020.

[RMP+20a] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Metalevel transformation of strategies. In *7th International Workshop on Rewriting Techniques for Program Transformation and Evaluation, WPTE 2020, Paris, France*, 2020. 10 pages.

[RMP+20b] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Simulating and model checking membrane systems using strategies in Maude. In *7th International Workshop on Rewriting Techniques for Program Transformation and Evaluation, WPTE 2020, Paris, France*, 2020. 10 pages.

[RMP+22a] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Metalevel transformation of strategies. *J. Log. Algebr. Methods Program.*, 124, 2022. 36 pages.

[RMP+22b] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Simulating and model checking membrane systems using strategies in Maude. *J. Log. Algebr. Methods Program.*, 124, 2022. 42 pages.

The different tools and examples included in this thesis are available at multiple repositories under the organization `github.com/fadoss` of the FaDoSS research group on GitHub:

- `maudesmc` contains the source code of the Maude LTL model checker extension for strategy-controlled systems explained in Section 4.3. It also includes the language module for checking branching-time properties using LTSmin described in Section 4.4, and the probabilistic extension of the strategy language in Chapter 5.
- `umaudemc` hosts the *unified Maude model-checking tool* `umaudemc` introduced in Section 4.4.
- `strat-examples` gathers all the examples appearing in this thesis and some additional ones. In the following, we will refer to this as the *example collection*.
- `maude-bindings` contains the code and documentation of the language bindings for Maude explained in Appendix A.

The Maude strategy language described in Chapter 3 is part of the official Maude 3 interpreter, available at `github.com/SRI-CSL/Maude`. All these resources and additional documentation can be accessed through the homepage of the Maude strategy language, `maude.ucm.es/strategies`.

Chapter 2

Preliminaries

This chapter reviews the state of the art of different topics involved in this thesis: strategies, model checking, rewriting logic and Maude, etc. Some definitions and notation are recalled for their use in the next chapters of this document.

2.1 Languages and automata over finite and infinite words

The theory of formal languages over finite words is a fundamental theory in computer science, but languages over infinite words [Sta97, PP04] are not so popular. However, the infinite word counterpart of regular languages has attracted significant attention as a resource for modeling the behavior of indefinitely-running and reactive systems, which are the main target of model checking. In this section, we review languages over finite and infinite words, which are later used for model checking and for the abstract representation of strategies, recalling some of their properties and definitions.

Finite words are sequences of symbols $s_1 \cdots s_n$ from a usually finite alphabet Σ, where ε denotes the empty word and $|w|$ the length of the word w, and languages are sets of these words $L \subseteq \Sigma^*$ that can be combined with some operations. They are sometimes described by grammars and abstract recognizing devices of different expressive power and computational properties, which give rise to multiple language classes organized in the Chomsky hierarchy in Table 2.1.

An *infinite word* on an alphabet Σ is an infinite sequence of elements of Σ or formally a function $w : \mathbb{N} \to \Sigma$. The set of all infinite words is written Σ^ω and the set of all finite and infinite words is denoted by $\Sigma^\infty := \Sigma^* \cup \Sigma^\omega$. An ω-language is a subset $L \subseteq \Sigma^\omega$ of infinite words, and an ∞-language may contain both finite and infinite words. Some typical operations can be defined on ∞-languages like union $L \cup M$, intersection $L \cap M$, concatenation $LM := \{vw : v \in L \cap \Sigma^*, w \in M\} \cup L \cap \Sigma^\omega$, power $L^n := L^{n-1} L$ with $L^0 := \{\varepsilon\}$, the Kleene star $L^* := \cup_{n \in \mathbb{N}} L^n$, and the infinite iteration $L^\omega := \{w_1 w_2 \cdots : w_k \in L \setminus \{\varepsilon\}\}$. Moreover, a natural prefix relation $\sqsubseteq$ can be defined on ∞-words, where $w \sqsubseteq v$ if $v \in \{w\}\Sigma^\infty$. Actually, the pair $(\Sigma^\infty, \sqsubseteq)$ is a chain-complete partially ordered set, i.e. $\sqsubseteq$ is a partial order and every completely

Type	Language class/grammar	Device class	Emptiness problem
3	Regular	Finite automata	decidable
2	Context-free	Pushdown automata	decidable
1	Context-sensitive	Linear bounded automata	undecidable
0	Recursively enumerable	Turing machines	undecidable

Table 2.1: Chomsky hierarchy of languages over finite words.

ordered subset has a supremum, the shortest word that has all the words of the chain as prefixes. A language L is *prefix-closed* if all the prefixes of every word in L are included in L. Conversely, a language L containing all words whose finite prefixes are in L, either independently or as prefixes of other words, is called *closed*.

Definition 2.1 (closed language). *A language $L \subseteq \Sigma^\infty$ is* closed *if*

$$\forall w \in \Sigma^\infty \quad (\forall v \in \Sigma^* \;\; v \sqsubseteq w \;\Rightarrow\; \exists\, w' \in L \;\; v \sqsubseteq w') \;\Rightarrow\; w \in L.$$

Any $w \in \Sigma^\infty$ in the conditions of the definition is called an accumulation point of L, and so L is closed if it contains all its accumulation points. We write $\overline{L}$ for the closure of L, the smallest closed language containing L, where its accumulation points have been added. We also say that a sequence $(w_n)_{n\in\mathbb{N}}$ converges to another word $w \in \Sigma^\infty$ if for every prefix $v \sqsubseteq w$ there is an $n \in \mathbb{N}$ such that $v \sqsubseteq w_m$ for all $m \geq n$. This topological terminology and notation is not gratuitous, since the previous notions have a precise and coincident topological sense when Σ^∞ is endowed with the appropriate topology to become a Cantor space. Its open sets are the languages $X\Sigma^\infty$ for $X \subseteq \Sigma^*$, sometimes called cylinder sets, which coincide with those of the Scott topology for the prefix relation $\sqsubseteq$, and can also be engendered by a distance defined as

$$d(w, v) \coloneqq \max\{0, \min\{2^{-n} : w_n \neq v_n \;\vee\; |w| < n \leq |v| \;\vee\; |v| < n \leq |w|\}\}.$$

Intuitively, the distance between two words decreases as the length of their common prefix increases. Infinite-word languages have largely been studied from a topological and descriptive set theory point of view, where (Σ^∞, d) is a so-called Polish space [PP04].[1]

The Chomsky hierarchy of finite word languages is also ported to the infinite word case, whose language and device names are prefixed by an ω, extending the ideas behind ω-regular languages [Fin14b]. This class is the infinite word version of regular languages and its recognizing device is the analogue of the finite automaton, the Büchi automaton. They were introduced by Julius Richard Büchi in 1962 to study a pure mathematical problem, the decidability of a restricted second-order arithmetic theory of the integers. A Büchi automaton $M = (Q, \Sigma, \delta, q_0, F)$ consists of a finite set Q of automaton states, an initial state q_0, a nondeterministic[2] transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$, and an acceptance condition F. A word w is accepted by M if there is a run $\pi = q_0 q_1 \cdots$ satisfying the acceptance condition such that $q_k \in \delta(q_{k-1}, w_k)$. Their only difference with finite automata is that they recognize infinite words, and consequently the acceptance condition F cannot refer to the final states where executions stop. Instead, $F \subseteq Q$ represents a subset of states that must occur infinitely often in the run in order to be accepted, i.e. $\inf(\pi) \cap F \neq \varnothing$ where $\inf(\pi) = \{q \in Q : q \text{ appears infinitely often in } \pi\}$. An acceptance condition of this form is called a Büchi condition, but there are other alternatives. For example, a run $\pi \in Q^\omega$ is accepting if

- Under a co-Büchi condition $F \subseteq Q$, $\inf(\pi) \cap F = \varnothing$.
- Under a Streett condition $F = \{(A_1, B_1), \ldots, (A_n, B_n)\} \subseteq Q^2$ of index n, if $\inf(\pi) \cap A_k = \varnothing$ or $\inf(\pi) \cap B_k \neq \varnothing$ for all $1 \leq k \leq n$.
- Under a Rabin acceptance condition F of index n, whose shape is that of a Streett condition, if $\inf(\pi) \cap A_k = \varnothing$ and $\inf(\pi) \cap B_k \neq \varnothing$ for some $1 \leq k \leq n$.
- Under a Muller acceptance condition $F \subseteq \mathcal{P}(Q)$, if $\inf(\pi) \in F$.

Nondeterministic automata are equally expressive under Büchi, Streett, Rabin, and Muller acceptance, but they are less expressive with co-Büchi conditions. Moreover, determinism does

[1] A Polish space is a separable completely metrizable topological space. More precisely, for (Σ^∞, d) to be separable and so Polish space, Σ must be countable.

[2] Unlike for finite automata, deterministic Büchi automata are less expressive than their nondeterministic equivalent.

not reduce the expressive power of co-Büchi, Street, Rabin, and Muller automata, as it does in the Büchi case.

Another way of describing ω-regular languages is by using ω-regular expressions (also known as ω-rational). They extend the classical regular expressions with an operator α^ω for the infinite iteration of a language.

$$\alpha ::= \varnothing \mid \varepsilon \mid s \mid \alpha\alpha \mid (\alpha \mid \alpha) \mid \alpha^* \mid \alpha^\omega$$

Expressions are given meaning recursively, with $L(\varnothing) = \varnothing$, $L(\varepsilon) = \{\varepsilon\}$, $L(s) = \{s\}$ for $s \in \Sigma$, $L(\alpha\beta) = L(\alpha)L(\beta)$, $L(\alpha \mid \beta) = L(\alpha) \cup L(\beta)$, $L(\alpha^*) = L(\alpha)^*$ and $L(\alpha^\omega) = L(\alpha)^\omega$. The lefthand side operand of concatenations and derived operators is usually required to denote a finite word language by excluding the ω operator. Every ω-regular language can be described by an ω-regular expression.

Turing machines and recursively enumerable languages do also have their replica for infinite words. An ω-Turing machine is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q is a finite set of states, Σ is the finite input alphabet, Γ is a finite tape alphabet with $\Sigma \subseteq \Gamma$, F is a set of states to define a Büchi-like condition, q_0 is an initial state, and $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, S\})$ is a nondeterministic transition function, where the letters indicate whether the head may move left L, right R, or stay in place S. Like for Büchi automata, an ω-Turing machine is a Turing machine whose set of final states becomes an acceptance condition. However, other restrictions are imposed on the tape positions visited and the movements of the head. For example, in the type 2 semantics of ω-Turing machines, accepting runs should be complete, i.e. they should visit every position of the tape. A run r on M for a word w is an infinite sequence of configurations $r = (q_i, w_i, j_i)_{i=1}^\infty$ with $r_1 = (q_0, w, 0)$ and $r_{k+1} = (q_{k+1}, w_k[j_k/s], j_k + m)$ if $(q_{k+1}, s, m) \in \delta(q_k, (w_k)_{j_k})$ where m is -1 for L, 1 for R and 0 for S. A word w is accepted by the Turing machine M if there is a complete run such that $q_k \in F$ for infinitely many k. Other acceptance conditions can be considered, like for Büchi automata. More details can be found in [Sta97, Fin14a].[3]

2.2 Strategies

Strategies, tactics, or policies are words in the common language whose meaning is probably known to everyone. Most dictionaries agree in defining *strategy* as

1. *a detailed plan for achieving long-term or overall goals under uncertain conditions*, or simply
2. *a way of doing or dealing with something*.

This word is borrowed in many languages from the ancient Greek *στρατηγία*, *the office of a general*, who is in charge of the overall planning of the military operations. One of the most famous treatises about strategies is *The Art of War* written by the Chinese general Sun Tzu around twenty-five centuries ago, but strategies have been proposed and developed for the most varied and peaceful activities.

In Computer Science, strategies are widespread and play a similar role. They can be found since its origins in the λ-calculus [Bar14], in functional programming to evaluate terms, in operating systems and web services to distribute the resources and the work among the multiple processing units, in inference and deductive procedures to guide them towards the expected goal, in games to win or obtain the best possible result, and so on. The following sections introduce some general formalizations of strategies that we will find convenient in this thesis, and some strategy languages used to describe them in practice in the context of formal specification. Strategies are conveniently formalized in state and transition systems, which are explained first.

[3] In this thesis, ω-Turing machines are only used in Proposition 3.2 to prove the Turing completeness of the Maude strategy language.

2.2.1 State and transition systems

Transition systems are extensively used as the basis of the formal modeling of hardware devices, communication protocols, reactive systems, computer programs, games, languages, etc. Since they are collections of abstract states connected by abstract transitions, dynamic systems can be represented in the simplest form.

Definition 2.2 (transition systems). *Given a set S of states, a* transition system *(TS) or* abstract reduction system *(ARS) $\mathcal{A} = (S, R)$ consists of a binary relation $R \subseteq S \times S$ on the states. A* labeled transition system *(LTS) $\mathcal{A} = (S, A, R)$ consists of a set of actions A and a labeled relation $R \subseteq S \times A \times S$ on the states.*

We will usually refer to labeled transition systems when actions can be relevant, and to unlabeled transition systems otherwise. However, both notions are almost interchangeable, since an unlabeled (S, R) can be seen as a labeled $(S, \{\tau\}, R')$ with $(s, \tau, s') \in R'$ iff $(s, s') \in R$ for some fake action τ. Conversely, a labeled (S, A, R) can be represented by an unlabeled $(A \times S, R')$ by pushing the label to the target state, where $((a, s), (a', s')) \in R'$ if $(s, a', s') \in R$.

Transition relations are commonly denoted by arrows $\to$, and we write $s \to s'$ for $(s, s') \in R$ or $(s, a, s') \in R$, and $s \to^a s'$ for $(s, a, s') \in R$. These are called execution or reduction steps, and any $s' \in S$ such that $s \to s'$ is called a *successor* of s. The *executions* of a system are the finite or infinite sequences of states $s_0 \to s_1 \to \cdots \to s_n$ or $s_0 \to^{a_1} s_1 \to^{a_2} \cdots \to^{a_n} s_n$ linked by the relation. We represent them as finite $s_0 s_1 \cdots s_n$ or infinite $s_0 s_1 \cdots$ words, and designate by $\Gamma^*_{\mathcal{A}} \subseteq S^*$, $\Gamma^\omega_{\mathcal{A}} \subseteq S^\omega$, and $\Gamma_{\mathcal{A}} \subseteq S^\infty$ the set of all finite and infinite executions, and the union of both, respectively. A subscript $s \in S$ is added to these sets $\Gamma_{\mathcal{A},s}$ to indicate that only executions starting from this state are included. Executions on labeled transition systems are represented as zipper $(S \cup A)$-words $s_0 a_1 s_1 \cdots a_n s_n$, although seeing them as word of pairs $s_0\,(a_1, s_1) \cdots (a_n, s_n)$ in $S \times (A \times S)^\infty$ is also possible. We will write $\to^n$ for n aggregated steps of the relation, $\to^* = \cup_{n\in\mathbb{N}} \to^n$ for its transitive and reflexive closure, and $\to^+ = \cup_{n\geq 1} \to^n$ for its transitive closure. A state s is *irreducible* if there is no s' such that $s \to s'$, and we say that s' is a *normal form* of s and write $s \to^! s'$ when $s \to^* s'$ and s' is irreducible. A transition system $(S, \to)$ is *confluent* or has the Church-Rosser property if for all $u, u_1, u_2 \in S$ such that $u \to^* u_1$ and $u \to^* u_2$, there is a $v \in S$ such that $u_1 \to^* v$ and $u_2 \to^* v$, it is *terminating* if there is no infinite execution, and it is *convergent* if it is both confluent and terminating. Confluence guarantees that normal forms are unique, while termination ensures that they exist.

Working with finite executions is sometimes problematic and most temporal logics used in formal verification, including those we will describe in Section 2.3, only consider nonterminating executions for simplicity [Pnu77]. Some or all finite executions in $\Gamma^*_{\mathcal{A}}$ may be meaningless incomplete executions that should not be considered. Anyhow, the system being modeled may present both finite and infinite executions. In this case, all of them are often considered infinite ones by the so-called *stuttering extension*, which consists of repeating the last state of finite executions forever, as if the system continues in an idle state. This extension can be implemented in the transition system by adding self-loops to deadlock states. However, this procedure is not selective with finite traces and does not support the possibility of stopping in the middle of a longer execution. Hence, we introduce the following stuttering extension given a subset of states where the transition system may halt.

Definition 2.3 (stuttering extension). *Given a language $L \subseteq S^\infty$, its stuttering extension is $L \cap S^\omega \cup \{s_1 \cdots s_n s_n \cdots : s_1 \cdots s_n \in L\}$. Moreover, given a subset $H \subseteq S$ of halting states, we define the stuttering extension of $\mathcal{A}$ with respect to H as*

$$\mathcal{A}_H = (S \times \{0\} \cup H \times \{1\}, \to_H)$$

where $(s, 0) \to_H (s', 0)$ iff $s \to s'$ for all $s, s' \in S$, and $(s, k) \to_H (s, 1)$ for all $s \in H$ and $k \in \{0, 1\}$.

The halting states are duplicated in $\mathcal{A}_H$ and a self-loop is added to the copy in order to avoid introducing these stuttering steps in the middle of other executions. This construction can be simplified, since deadlock states do not have successors and this undesired situation cannot happen. An additional action can be introduced in the labeled case for these self-loops. This construct with particular improvements will be used in Section 3.5.

2.2.2 Strategies

In the context of (possibly labeled) transition systems $\mathcal{A} = (S, A, \to)$, strategies have been formalized from different points of view [BCD⁺09]:

1. As functions $F : S \to \mathcal{P}(S)$ that identify states $s' \in F(s)$ yielded by finite controlled executions of the system from any initial state s, which must satisfy $s \to^* s'$. This notion has been used for the λ-calculus [Bar14] and we describe the Maude strategy language likewise in Section 3.4, but it lacks information about the nonterminating executions and about the intermediate execution steps.

2. A subgraph of the transition system $\mathcal{A}$ having the same set of normal forms [Ter03]. However, these strategies can only decide their next step based on the current one, while we are interested in memoryful and potentially more complex control mechanisms.

3. As a partial function $\lambda : (S \cup A)^+ \rightsquigarrow \mathcal{P}(A \times S)$ that selects the possible next steps to continue an execution based on its history, where $(a, s') \in \lambda(ws)$ must always satisfy $s \to^a s'$. The same definition without actions is valid for unlabeled systems. These functions are called *intensional strategies*, and they are extensively used in games and in the strategic logics mentioned in Section 2.6.

4. As a subset $E \subseteq \Gamma_\mathcal{A}$ of allowed executions of $\mathcal{A}$, which are called *abstract* or *extensional strategies* [KKK08].

Among all these definitions, we will stick with the last two, using the most appropriate one on each situation. Intensional strategies are less expressive than extensional strategies. In fact, any intensional strategy can be seen as an extensional one by taking

$$E(\lambda) = \{w \in (S \cup A)^\infty : (a_{k+1}, w_{k+1}) \in \lambda(w_0 \cdots a_k w_k), 0 \leq k < |w|/2\},$$

but the converse is not true. Even if an intensional strategy λ_E can also be defined from an extensional strategy E,

$$\lambda_E(w) = \{(a, s) \in A \times S : wasw' \in E, w' \in (S \cup A)^\infty\},$$

some information is lost and the inclusion $E \subseteq E(\lambda_E)$ could be strict. On the one hand, extensional strategies can be arbitrarily selective with finite executions, while intensional strategies cannot discriminate between a complete finite execution and their prefixes.[4] However, this will not be a problem for model checking, since we will usually consider nonterminating executions only by the stuttering extension of Definition 2.3 or logics that cannot distinguish that difference. On the other hand, $E(\lambda)$ is by definition a closed language as defined in Section 2.1 while an arbitrary extensional strategy might not be [BCD⁺09]. For example, these may allow executions of the form $a^n b^\omega$ for all $n \geq 0$ but not a^ω, while those cannot achieve that since the step that stays in a must always be allowed. This means that fairness restrictions cannot be represented in the strategy, but this is a reasonable assumption for practical executable strategies, and those restrictions can be treated apart. In case E only contains infinite executions, the correct equation would be $E(\lambda_E) = \overline{E}$.

In conclusion, we will mainly use the definition of extensional strategies for its simplicity and expressiveness, but sometimes intensional strategies will be more convenient, since they

[4]In our first paper [RMP⁺19a], we define intensional strategies $\lambda : S^+ \rightsquigarrow \mathcal{P}(S \cup \{●\})$ with an additional stop sign ● to mark allowed finite executions, but we no longer follow this definition since it was neither standard nor practical.

filter the possible next steps locally instead of globally. We will also say that an extensional strategy E is intensional if it is prefix-closed and closed, or simply closed. Actions will only be considered in strategies when dealing with labeled transition systems, but notice that their presence is not superfluous even if we only look at states, since they are part of the execution history and may condition future steps. The pair $(\mathcal{A}, E)$ will be called a strategy-controlled system.

2.2.3 Strategy languages

The abstract definitions of the previous section are convenient for the mathematical manipulation of strategies, but they are not used to describe strategies in practice. In term rewriting systems and λ-calculus, strategies are usually expressed with natural language phrases like "reduce the outer leftmost redex repeatedly" or "apply the rule r_1 before r_2". In order to express more complex strategies and have them executed by a machine, they started to be represented as programs in some strategy languages for rule-based formal specification, program transformation, and other applications. In the Maude specification language, these programs can be written at the metalevel, as we will see in Section 2.5, or using the Maude strategy language, thoroughly described in Chapter 3. This language is neither the first nor the unique in this kind, and we mention some other relevant examples:

- ELAN [BKK+01] (1993-2006) was the pioneer of strategy languages, developed at the LORIA group of the Université de Lorraine, CNRS, and INRIA. Like Maude, this is a rule-based language of the OBJ family, but for ELAN both rules and strategies were first class concepts from the beginning. The language included an interpreter, a compiler, and a standard library, and its development stopped in 2006.

- Stratego [BKV+08] (1998-) is oriented to control rule-based program transformations and includes a vast repertory of strategy combinators. It is now distributed as part of the Spoofax Language Workbench [WKV14], a platform for the development of textual, usually domain-specific, programming languages.

- Tom [BBK+07] (2001-) is a rewriting extension on Java that allows defining signatures, rules, and strategies inside the Java code and interoperate with them. Like ELAN, it is also developed at the LORIA group.

- ρLog [MK06] (2004-) is a strategy language for the rule-based engine of the Mathematica computing system, in which it is integrated as a plugin. It is developed by Mircea Marin, Temur Kutsia and Besik Dundua.

- Porgy [FKP19] (2009-) is a language for strategic graph (instead of term) rewriting included in a visual platform for modeling, simulating, and analyzing graph rewriting systems. Its authors are Maribel Fernández, Hélène Kirchner, who was also involved in ELAN, and Bruno Pinaud.

A detailed comparison of the syntax and semantics of ELAN, Stratego, Tom, ρLog, and the Maude strategy language is given in Section 3.8.

2.2.4 Execution trees

Since model checking of branching-time properties will be discussed, we need to pay attention to the concept of *execution tree* of a transition system from an initial state. The classical presentations of branching-time logics usually avoid giving a formal definition [CE81, EH86], with the implicit understanding that this is the informal infinite tree generated from the initial state by recursively adding as children all the successors of a state, as illustrated in Figure 2.1. Under

(a)
$$\begin{tikzcd}
& 2 \arrow[dl] \arrow[dd] \\
1 \arrow[ur] & \\
& 3 \arrow[ul] \arrow[loop right]
\end{tikzcd}$$

(b)
$$\begin{tikzcd}
& & 1 \arrow[r] & 2 & \cdots \\
1 \arrow[r] & 2 \arrow[ur] \arrow[dr] & & 1 & \cdots \\
& & 3 \arrow[ur] \arrow[dr] & & \\
& & & 3 & \cdots
\end{tikzcd}$$

(c)
$$\begin{tikzcd}
& & 1 \arrow[r] & 0 & \cdots \\
1 \arrow[r] & 0 \arrow[ur] \arrow[dr] & & 1 & \cdots \\
& & 1 \arrow[ur] \arrow[dr] & & \\
& & & 1 & \cdots
\end{tikzcd}$$

Figure 2.1: A transition system (a), its execution tree (b), and a projection by $f(n) = n \bmod 2$.

the control of a strategy, the execution tree is pruned since some executions are not allowed, so it becomes a subtree of the original execution tree.

From a graph-theoretic point of view, the execution tree from a given initial state $r \in S$ in an LTS $\mathcal{A} = (S, A, \to)$ is the A-labeled graph $(\Gamma^*_{\mathcal{A},r}, \{(ws, a, wsas') : s \to^a s', ws \in \Gamma^*_{\mathcal{A},r}\})$. This graph is a rooted tree, since it does not contain cycles and r is its root. Each vertex ws holds not only the current execution state s but the whole execution history. An intensional strategy λ can be then seen as a subtree of this execution tree, namely $(\Gamma^*_{\mathcal{A},r}, \{(w, a, was) : (a, s) \in \lambda(w), w \in \Gamma^*_{\mathcal{A},r}\})$. In the unlabeled case, vertices are only words on states, and the graph is an unlabeled one. An arbitrary extensional strategy cannot be faithfully seen as a tree without adding halting marks or acceptance conditions to their states, but we will not need it.

In the context of model checking, it is more common to see trees formalized as S-*labeled trees* (T, V) from descriptive set theory. They consist of a subset $T \subseteq \mathbb{N}^*$ and a labeling function $V : T \to S$, its nodes are the words $k_1 \cdots k_n \in T$, and their children are $k_1 \cdots k_n k \in T$ for $0 \leq k < d$ where d is the degree of $k_1 \cdots k_n$. The graph-theoretic trees of the previous paragraph can be seen as S-labeled trees by giving an arbitrary numbering to their children, and the opposite conversion is also possible. One of the advantages of this concept is that nodes can be relabeled seamlessly by taking $(T, f \circ V)$ for any $f : S \to S'$, and this makes sense when model checking, since states are seen as the atomic propositions they satisfy. Hence, in the following, the meaning of the execution tree allowed by a strategy λ will be any of these equivalent definitions.

2.3 Model checking

Model checking [CHV+18] is an automated verification method for proving or refuting properties of the dynamic behavior of a model. It is not a single concrete verification technique, but an umbrella term comprising many heterogeneous but related instances where models and properties are expressed in different ways, and different algorithms and space-reduction techniques are used. However, its models are always variations of state and transition systems, whose executions are explored more or less exhaustively. This allows a great degree of automation, but also involves a combinatorial explosion problem. Model checking is widely used in industry for the verification of hardware and software.

In the classical framework, models are state and transition systems (see Section 2.2) whose states are annotated with *atomic propositions*, in terms of which the desired properties are expressed. Such construct receives the name of *Kripke structure* $\mathcal{K} = (S, \to, I, AP, \ell)$ where $(S, \to)$ is a labeled or unlabeled transition system, I is a set of initial states, AP is the set of atomic propositions, and $\ell : S \to \mathcal{P}(AP)$ is the labeling function that declares which atomic properties are satisfied in each state. It is usually assumed that the transition relation $\to$ is *total*, i.e. that every state has a successor, to only consider infinite executions. However, if it were not, the stuttering extension explained in Section 2.2 could be applied.

Properties are often expressed using *temporal logics* that extend a propositional logic whose basic predicates are the atomic propositions of the model with temporal operators to describe how they should occur in time. Temporal logics and properties are usually classified as [Lam80]:

- *Linear-time* properties, which are universal properties satisfied by every possible execution of the system. In other words, time is seen as a sequence of states where the next step is already determined. The main representative is Linear-time Temporal Logic [Pnu77] (LTL) and its multiple extensions, but properties can also be expressed as an automaton, like the *never claims* of the Spin model checker [Hol+21].
- *Branching-time* properties reason about the whole execution tree, where multiple futures can be available at any moment. Well-known examples are Computational Tree Logic (CTL) [CE81], and the more general CTL* that includes both LTL and CTL.

Another classification distinguishes between *state-based* and *action-based* logics [DV90], depending on whether formulae refer to properties of the model states or of their actions or transitions. The first class includes all the logics we have mentioned up to now. This distinction is not essential since transition labels can be pushed to the states and vice versa, as commented in Section 2.2. Moreover, both kinds of properties can be considered together, like in μ-calculus [Koz83] and the Temporal Logic of Rewriting [Mes08] (TLR).

The semantics of temporal logics is usually defined by means of satisfaction relations $\mathcal{K}, s \vDash \varphi$. In the case of linear-time properties, the satisfaction of a formula φ is reduced to its satisfaction $\mathcal{K}, \ell(\pi) \vDash \varphi$ for all the executions $\pi \in \Gamma^{\omega}_{\mathcal{K},s}$ of the system. We could say that a linear-time property accepts or rejects executions, while a branching-time one does so with trees. The model-checking problem is deciding whether a model satisfies a given property. The sequences of sets of atomic propositions being satisfied by the model executions, in other words, the projections of the executions of $\mathcal{K}$ by ℓ, are called the *propositional traces* of $\mathcal{K}$. Linear-time properties can be characterized as subsets $P \subseteq \mathcal{P}(AP)^\infty$ of propositional traces or zipped $P \subseteq (\mathcal{P}(AP) \cup A)^\infty$ for the labeled case. Similarly, branching-time properties could be expressed as sets Q of $\mathcal{P}(AP)$-labeled trees, although this characterization of branching-time properties is not frequent [KVW00].

In addition to what we have explained here, model checking is also applied to real-time systems, where time is a continuous measure instead of the discrete sequence of the execution steps, to probabilistic systems by probabilistic and statistical model checking, etc. A good and recent monograph on model checking is [CHV+18].

We finish this global discussion about model checking by introducing the notion of bisimulation between Kripke structures [CHV+18; § 26.3.1]. When two Kripke structures are related likewise, both satisfy exactly the same temporal formulae for many logics, including CTL* and μ-calculus, which we will introduce in the following sections.

Definition 2.4 (bisimulation). *Given two (labeled) Kripke structures $\mathcal{K}_1 = (S_1, A, R_1, I_1, AP, \ell_1)$ and $\mathcal{K}_2 = (S_2, A, R_2, I_2, AP, \ell_2)$, a bisimulation is a relation $B \subseteq S_1 \times S_2$ such that if $(s_1, s_2) \in B$ then*

- $\ell_1(s_1) = \ell_2(s_2)$,
- *for every action a and state s_1' such that $(s_1, a, s_1') \in R_1$, there is some $s_2' \in S_2$ such that $(s_2, a, s_2') \in R_2$ and $(s_1', s_2') \in B$.*
- *the symmetrical condition, with s_2 and s_2' in the roles of s_1 and s_1'.*

If the Kripke structures are not labeled, the same definition is valid, as if we see them as labeled transition systems with a single action. Two states $s_1 \in S_1$ and $s_2 \in S_2$ are *bisimilar* if there is a bisimulation relation B satisfying $(s_1, s_2) \in B$. Two Kripke structures $\mathcal{K}_1$ and $\mathcal{K}_2$ as above are *bisimilar* if for every initial state $s_1 \in I_1$ there is a bisimilar initial state $s_2 \in I_2$ and vice versa.

2.3.1 CTL*

CTL* [EH86] is a branching-time temporal logic that extends both LTL and CTL. Its formulae are constructed over the atomic propositions of a Kripke structure, combined with some temporal operators and path quantifiers.

$$\begin{aligned}\Phi &::= \bot \mid \top \mid p \mid \neg\,\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathbf{A}\,\phi \mid \mathbf{E}\,\phi \\ \phi &::= \Phi \mid \neg\,\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \bigcirc\,\phi \mid \Diamond\,\phi \mid \Box\,\phi \mid \phi\,\mathbf{U}\,\phi\end{aligned}$$

Terms built from ϕ are called *path formulae* and they describe properties of fixed execution paths: $\bigcirc\,\phi$ tells that the property ϕ is satisfied in the next state of the path, $\Diamond\,\phi$ and $\Box\,\phi$ say that ϕ is satisfied in some or all states of the path respectively, and $\phi_1\,\mathbf{U}\,\phi_2$ claims that ϕ_2 is satisfied in some state and ϕ_1 holds until then. Terms under the Φ symbol are called *state formulae* since they refer to a state of the system, to the atomic properties $p \in AP$ it satisfies, and to the paths leaving from it, quantified either universally $\mathbf{A}\,\phi$ or existentially $\mathbf{E}\,\phi$.

The semantics of CTL* formulae is given by a satisfaction relation on states $\mathcal{K}, s \vDash \Phi$ and on paths $\mathcal{K}, \pi \vDash \phi$. We will not write its definition for all operators above, since some of them can be defined in terms of the others, like $\phi_1 \vee \phi_2 = \neg(\neg\,\phi_1 \wedge \neg\,\phi_2)$, $\Diamond\,\phi = \top\,\mathbf{U}\,\phi$, $\Box\,\phi = \neg\Diamond\neg\,\phi$, and $\mathbf{A}\,\phi = \neg\,\mathbf{E}\,\neg\,\phi$. We write $\pi^k := (\pi_{k+n})_{n=0}^{\infty}$ for the suffix of π starting from the k-th element.

1.	$\mathcal{K}, s \vDash p$	iff $p \in \ell(s)$
2.	$\mathcal{K}, s \vDash \neg\,\Phi$	iff $\mathcal{K}, s \nvDash \Phi$
3.	$\mathcal{K}, s \vDash \Phi_1 \wedge \Phi_2$	iff $\mathcal{K}, s \vDash \Phi_1$ and $\mathcal{K}, s \vDash \Phi_2$
4.	$\mathcal{K}, s \vDash \mathbf{E}\,\phi$	iff $\exists\,\pi \in \Gamma_s \quad \mathcal{K}, \pi \vDash \phi$
5.	$\mathcal{K}, \pi \vDash \Phi$	iff $\mathcal{K}, \pi_0 \vDash \Phi$
6.	$\mathcal{K}, \pi \vDash \neg\,\varphi$	iff $\mathcal{K}, \pi \nvDash \varphi$
7.	$\mathcal{K}, \pi \vDash \varphi_1 \wedge \varphi_2$	iff $\mathcal{K}, \pi \vDash \varphi_1$ and $\mathcal{K}, \pi \vDash \varphi_2$
8.	$\mathcal{K}, \pi \vDash \bigcirc\,\varphi$	iff $\mathcal{K}, \pi^1 \vDash \varphi$
9.	$\mathcal{K}, \pi \vDash \varphi_1\,\mathbf{U}\,\varphi_2$	iff $\exists\,n \geq 0 \;\; \mathcal{K}, \pi^n \vDash \varphi_2 \;\wedge\; \forall\,0 \leq k < n \;\; \mathcal{K}, \pi^k \vDash \varphi_1$

Linear-time Temporal Logic (LTL) is the subset of CTL* formulae of the form $\mathbf{A}\,\phi$ where ϕ does not contain path quantifiers, and the initial $\mathbf{A}$ is left implicit. Computational Tree Logic (CTL) is the subset in which every path operator is preceded by a quantifier. Historically, these logics have not been defined as subsets of CTL*. In fact, LTL was first proposed in 1977 by Amir Pnueli [Pnu77], and CTL in 1981 by Edmund M. Clarke and E. Allen Emerson [CE81], while CTL* was first defined in 1986 by Emerson and Joseph Y. Halpern. Well-known model checkers for these logics are Spin [Hol+21] for LTL, and NuSMV [CCG+02] for LTL and CTL. Model checkers for CTL* are rare, but this logic is supported by LTSmin [KLM+15].

The satisfaction of a CTL* property is invariant by bisimulation, i.e. two bisimilar Kripke structures satisfy the same formulae. In fact, satisfying the same formulae and being bisimilar are equivalent conditions as the following theorem states:

Theorem 2.1 ([BK08; Theorem 7.20])**.** *Two states, s_1 of $\mathcal{K}_1$ and s_2 of $\mathcal{K}_2$, are bisimilar iff $\mathcal{K}_1, s_1 \vDash \varphi \iff \mathcal{K}_2, s_2 \vDash \varphi$ for all CTL* (or for all CTL) formulae φ.*

2.3.2 LTL model checking

Since LTL is a sublogic of CTL*, its syntax and semantics have already been given in Section 2.3.1. In this section, we introduce the typical implementation of explicit-state LTL model checking, the so-called *automata-theoretic approach* [CGP99], used in the Maude LTL model checker and our extension. This method is based on Büchi automata algorithms and the fact that the language $L(\varphi) = \{\rho \in \mathcal{P}(AP)^{\omega} : \mathcal{K}, \rho \vDash \varphi\}$ of propositional traces described by an LTL formula φ is an ω-regular language [Pnu77].

The model-checking problem is equivalent to the language inclusion problem $\ell(\Gamma^\omega_\mathcal{K}) \subseteq L(\varphi)$ or equivalently to deciding whether $\ell(\Gamma^\omega_\mathcal{K}) \cap L(\neg\varphi) = \varnothing$. Since $\ell(\Gamma^\omega_\mathcal{K})$ is also an ω-regular language, the problem is decidable and PSPACE-complete by the results from automata theory on infinite words. Hence, model checking can be reduced to the following steps:

1. Generating a Büchi automaton B for $\neg\varphi$. The number of its states can be exponential on the size of the formula, but this is not frequent in practice. This translation usually starts by generating a very weak alternating automaton, then a generalized Büchi automaton, and finally the Büchi automaton [GO01] after some intermediate optimizations.

2. Generating an automaton for the model. Any transition system can be seen as a Büchi automaton with trivial acceptance condition $M = (S \cup \{\iota\}, \mathcal{P}(AP), \delta, \iota, S \cup \{\iota\})$ that recognizes $\Gamma^\omega_\mathcal{K}$, where $\delta(\iota, P) = \{s \in I : \ell(s) = P\}$, and $\delta(s, P) = \{s' \in S : s \to s' \wedge \ell(s') = P\}$.

3. Calculating the intersection $L(B) \cap L(M)$, with the (synchronous) product automaton $B \times M$.

4. Checking whether the intersection is empty, using a nested depth-first search [HPY97] that yields a counterexample in case it is not.

The last three steps can be performed simultaneously, generating the model automaton as required by the property, on the fly.

2.3.3 μ-calculus

Modal μ-calculus [Koz83] is an extension of Hennessy-Milner logic [HM80] with least μ and greatest ν fixed-point operators. It can be used to express edge-aware properties on labeled transition systems using two modalities, $[a]\,\varphi$ that asserts that all states reachable by an action a satisfy φ, and $\langle a\rangle\,\varphi$ which claims the existence of a successor by a that satisfies φ. Formulae may contain variables Z that must be bound by fixed-point operators.

$$\varphi \ ::= \ \bot \mid \top \mid p \mid Z \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid [a]\,\varphi \mid \langle a\rangle\,\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

The value of a μ-calculus formula is the set of states in which it holds, written $[\![\phi]\!]_\eta$, where $\eta : \mathrm{Var} \to \mathcal{P}(S)$ is an assignment of values to the free variables that may appear in nested formulae. Formally, the semantics is defined as follows:

1. $[\![p]\!]_\eta \quad = \{s \in S : p \in \ell(s)\}$
2. $[\![\neg\varphi]\!]_\eta \quad = S \setminus [\![\varphi]\!]_\eta$
3. $[\![\varphi_1 \wedge \varphi_2]\!]_\eta \quad = [\![\varphi_1]\!]_\eta \cap [\![\varphi_2]\!]_\eta$
4. $[\![Z]\!]_\eta \quad = \eta(Z)$
5. $[\![\langle a\rangle\,\varphi]\!]_\eta \quad = \{s \in S : \exists\, s' \in [\![\varphi]\!]_\eta \quad (s, a, s') \in R\}$
6. $[\![\nu Z.\varphi]\!]_\eta \quad = \bigcup\,\{U \subseteq S : U \subseteq [\![\varphi]\!]_{\eta[Z/U]}\}$

where $\eta[Z/U]$ is the function η with its value for the variable Z replaced by U. We have only defined the semantics of a subset of the logic constructors, since again some can be defined in terms of others: $[a]\,\varphi = \neg\langle a\rangle\neg\varphi$ and $\mu Z.\varphi = \neg(\nu Z.\neg\varphi)$. We sometimes write multiple actions $C \subseteq A$ in the modalities meaning $[C]\,\varphi = \bigwedge_{a\in C}[a]\,\varphi$ and $\langle C\rangle\,\varphi = \bigvee_{a\in C}\langle a\rangle\,\varphi$. In the extreme case where $C = A$, it is common to see $[\cdot]\,\varphi = [A]\,\varphi$ and $\langle\cdot\rangle\,\varphi = \langle A\rangle\,\varphi$. We say that a μ-calculus formula is a *sentence* if it does not contain free variables, and in this case, the assignment η is irrelevant and usually omitted.

For the fixed points to be well-defined, the function $U \mapsto [\![\varphi]\!]_{\eta[Z/U]}$ should be monotone, and so variables must appear under an even number of negations. This logic is more expressive but probably less intuitive than CTL*. For example, $\mathbf{A} \diamond p$ is $\mu Z.(p \vee [\cdot] Z)$ and $\mathbf{E} \square p$ is $\nu Z.(p \wedge \langle\cdot\rangle Z)$. The exact complexity of model checking is an open problem, but it is known to be in NP, co-NP, and P-hard [CHV⁺18]. As well as the previous logics, μ-calculus is invariant by bisimulation.

Theorem 2.2 ([CHV⁺18; Theorem 6:10]). *If a state s_1 of $\mathcal{K}_1$ is bisimilar to a state s_2 of $\mathcal{K}_2$ then for every μ-calculus sentence φ: $s_1 \in [\![\varphi]\!]_{\mathcal{K}_1}$ iff $s_2 \in [\![\varphi]\!]_{\mathcal{K}_2}$.*

Not being as widespread as CTL and LTL, there are model checkers for the μ-calculus like mCRL2 [BGK⁺19], LTSmin [KLM⁺15], and TAPAs [CDL⁺08].

2.3.4 A μ-calculus model-checking algorithm

Checking a μ-calculus property is equivalent to solving a parity game where a player tries to prove and another tries to refute the property [BW18], and this equivalence gives rise to practical model-checking algorithms. A parity game is a graph whose nodes are marked with integer numbers and belong to any of two players, say *zero* and *one*. The turn is of the owner of the current node, who must choose the next step and loses the game if no move is possible. When plays are infinite, the *parity* adjective becomes meaningful, since the winner is the player whose parity coincides with that of the highest node mark repeated infinitely often in the play.

Definition 2.5 (parity game). *A* parity game $G = (V, V_0, V_1, E, \Omega)$ *is a transition system* (V, E) *where states are partitioned in two sets $V = V_0 \cup V_1$, and a ranking function $\Omega : V \to \mathbb{N}$. A* play *in G is an execution in (V, E), and player k wins G from a state s_0 if there is a finite play $s_0 w s$ such that $s \in V_{1-k}$ is irreducible, or there is an infinite play $s_0 w$ such that*

$$\limsup\ \Omega(s_0 w) = \inf_{n \in \mathbb{N}} \sup_{m \geq n} \Omega((s_0 w)_m)$$

coincides with k modulo two.

It can be proved that from each state one and only one of the players wins the game. Moreover, winning is equivalent to having a winning strategy from that position, which can always be positional and deterministic $\lambda : V_k \to V$.[5]

The construction of a game from a Kripke structure $\mathcal{K}$ and a μ-calculus formula φ is very intuitive and we will briefly describe it here. We assume that player zero tries to prove φ on $\mathcal{K}$ and player one tries to refute it. The states of the game consist of a state in $\mathcal{K}$ and a subformula of φ, being (s_0, φ) the initial one for the initial state s_0 of the model. For example, $(s, \top)$ must not have successors and belong to player one, so that refutation is doomed to failure, and the same happens with (s, p) when $p \in \ell(s)$. Similarly, $(s, \varphi_1 \wedge \varphi_2)$ should have (s, φ_1) and (s, φ_2) as successors and belong to player one again, so that it can choose the unsatisfiable formula in the conjunction, if any, to refute it. On the contrary and dually, $(s, \varphi_1 \vee \varphi_2)$ should belong to player zero. The Kripke structure appears for states like $(s, \langle a \rangle\, \varphi)$ that belong to player zero and have a successor (s', φ) for every successor s' of s by the action a. Fixed points are the most complex case $(s, \mu Z.\varphi)$, which are linked recursively to $(s, \varphi[Z/\mu Z.\varphi])$ and whose owner does not really matter. However, the rank Ω, which can be defined as zero for all other formulae, is important in these latter constructs that deal with infinite computations. Greatest ν and least μ fixed points are assigned even and odd ranks respectively, in coherence with the less clear intuition that the first describe properties that must hold infinitely often while the second should be resolved finitely. An attribute of μ-calculus formulae called alternation depth is used to distinguish the rank of distinct fixed points. The complete details and proofs of this procedure are explained in [BW18], and we have implemented it for Maude models in Section 4.4.

[5]The concept of strategy is the same we have considered in Section 2.2, but here only the moves of one of the players are constrained. This is a natural variation for multiplayer games as we also see in Section 2.6.

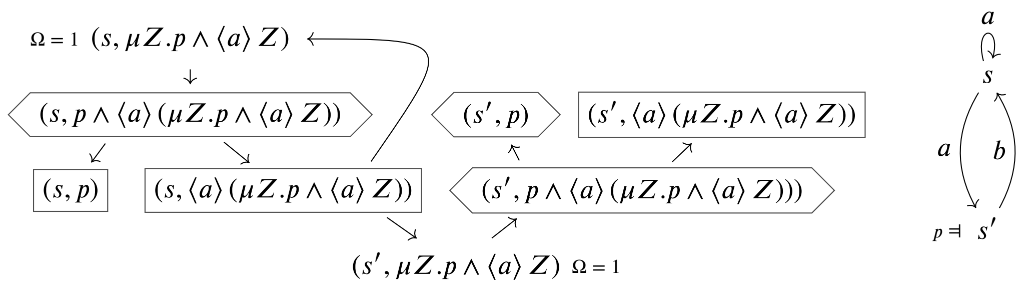


Figure 2.2: A parity game for the Kripke structure at the right.

Once the parity game is built, we need to solve it in order to check whether $s_0 \in [\![\varphi]\!]$ holds. One simple but rather efficient method is the Zielonka algorithm [Zie98]. It is a recursive algorithm based on the concept of attractor, which is the set $\text{attr}_G(U, k)$ of states in V in which player k can force the game to arrive to U.

$$\begin{aligned}
\text{attr}_G(U, k, 0) &= U \\
\text{attr}_G(U, k, i+1) &= \text{attr}_G(U, k, i) \cup \{v \in V_k \;:\; \exists\,(v, w) \in E \quad w \in \text{attr}_G(U, k, i)\} \\
&\qquad \cup \{v \in V_{1-k} \;:\; \forall\,(v, w) \in E \quad w \in \text{attr}_G(U, k, i)\} \\
\text{attr}_G(U, k) &= \textstyle\bigcup_{n\in\mathbb{N}} \text{attr}(U, k, n)
\end{aligned}$$

The auxiliary sets $\text{attr}_G(U, k, i)$ describe the states from which U can be reached in no more than i moves. Finite plays can be handled by simply calculating the attractors of the deadlock states $D_k \subseteq V_k$ of each player, since all states in $\text{attr}_G(D_{1-k}, k)$ are winning states for k. The main part of the algorithm assumes there are no deadlock states and deals with infinite executions by examining states of decreasing rank. Since the algorithm is recursive, the input data is a subgame $G' = (V', V_0', V_1', E', \Omega|_{V'})$ of G, starting from $G \setminus (D_0 \cup D_1)$. The result is a partition (W_0, W_1) of states in G' where each player wins.

1. Let p be the highest rank in G'. If $p = 0$, player zero wins in the whole V'.

2. Otherwise, let $k := p \bmod 2$ be the player that wins with this rank, and $U := \{v \in V' : \Omega(v) = p\}$ the subset of states where it is attained. Consider its attractor $A := \text{attr}_{G'}(U, k)$ and apply the algorithm on $G' \setminus A$, obtaining W_0' and W_1'. If $W_k' = V' \setminus A$, k wins in every state, so $W_k = V'$, $W_{1-k} = \emptyset$, and we have finished.

3. Otherwise, $1-k$ surely wins in $B := \text{attr}_{G'}(W_{1-k}, 1-k)$. In effect, k cannot escape from B to the complement of $G' \setminus A$ since A is an attractor. Hence, we should recursively solve the rest of the game $G' \setminus B$ to obtain W_0'' and W_1''. Finally, $W_k = W_k''$ and $W_{1-k} = W_{1-k}'' \cup B$.

This algorithm is extensively used to check branching-time properties in Section 4.4, because we have implemented it in the `umaudemc` tool, and it is also used by mCRL2 [BGK+19] and LTSmin [KLM+15] to check μ-calculus, CTL, and CTL* properties. There are other algorithms for solving parity games like SPM [Jur00] with lower asymptotic complexity, but Zielonka's often performs better in practice.

2.4 Rewriting logic

As explained in the introduction, rewriting logic [Mes92] is a formalism for expressing both concurrent computation and logical deduction in a general and natural way. Along with its implementation Maude, described in Section 2.5, it has been used to specify and verify many logics, models of concurrency, programming languages, hardware and software modeling languages, distributed algorithms, network and cryptographic protocols, real-time, cyber-physical and biological systems, etc. The survey [Mes12] is a valuable reference on rewriting logic, Maude and their applications, even though it was published almost a decade ago. In this section, we introduce the basic concepts behind this logic.

An algebraic specification [EM85] is based on a signature declaring some types and operation symbols. A *many-sorted signature* $\Sigma = (K, \Sigma, S)$ consists of a set of kinds K, a $K^* \times K$-indexed family $\Sigma = \{\Sigma_{k_1 \cdots k_n,k} : k_1 \cdots k_n \in K^*, k \in K\}$ of symbols, and a K-indexed family $S = \{S_k\}$ of sorts. An *order-sorted* signature is a many-sorted signature with a partial subsorting ordering $\leq_k$ on every set S_k. We say that a symbol $f \in \Sigma_{k_1 \cdots k_n,k}$ has arity n, range kind k, and receive n arguments of kinds $k_1, \dots, k_n$. Symbols with arity zero are called *constants*. A Σ-algebra is a set $A = \{A_k\}_{k \in K}$, a mapping from each $f \in \Sigma_{k_1 \cdots k_n,k}$ to a function $f^A : A_{k_1} \times \cdots \times A_{k_n} \to A_k$, and from each sort $s \in S_k$ to a set $A_s \subseteq A_k$. Moreover, if Σ is order-sorted and $s \leq_k s'$, then $A_s \subseteq A_{s'}$. A homomorphism between two Σ-algebras A and B is a function $h : A \to B$ such that $h(f^A(a_1, \dots, a_n)) = f^B(h(a_1), \dots, h(a_n))$ for every symbol f and all elements $a_1, \dots, a_n$. Indexed families are seen as the disjoint union of their sets when convenient.

Given a K-kinded set of variables $X = \{X_k\}$, the set of *terms* $T_{\Sigma,k}(X)$ of kind k in a signature Σ is defined inductively as the variables in X_k, and the syntactical elements $f(t_1, \dots, t_n)$ for all symbols $f \in \Sigma_{k_1,\dots,k_n,k}$ and all terms $t_i \in T_{\Sigma,k_i}(X)$ for $1 \leq i \leq n$. The set of all terms in the signature is written $T_\Sigma(X) = \{T_{\Sigma,k}(X)\}$ and it is a Σ-algebra. Terms without variables $T_\Sigma := T_\Sigma(\varnothing)$ are called *ground terms*. A *substitution* is a kind-preserving function $\sigma : X \to T_\Sigma(X)$ assigning terms to variables. It can be inductively extended $\overline{\sigma} : T_\Sigma(X) \to T_\Sigma(X)$ to replace all occurrences of the variables in a term, with $\overline{\sigma}(x) = \sigma(x)$ and $\overline{\sigma}(f(x_1, \dots, x_n)) = f(\overline{\sigma}(x_1), \dots, \overline{\sigma}(x_n))$. For any pair of substitutions σ_1, σ_2, we define their composition $(\sigma_2 \circ \sigma_1)(x) := \overline{\sigma_2}(\sigma_1(x))$, which satisfies $\overline{\sigma_2 \circ \sigma_1} = \overline{\sigma_2} \circ \overline{\sigma_1}$ in the usual functional sense. The line over the extension is usually omitted, and we also write $t[x_1 / t_1, \dots, x_n / t_n]$ to mean $\sigma(t)$ where $\sigma(x_i) = t_i$ and $\sigma(x) = x$ for $x \in X \setminus \{x_1, \dots, x_n\}$. Finally, a *position* $p \in \mathbb{N}^*$ in a term t indexes one of its subterms, written $t|_p$. The empty word ε selects the whole term t, and jp refers to the position p in the j-th argument of $t = f(t_1, \dots, t_n)$ if defined. We use $t[p / u]$ to denote the result of replacing the position p of t by u.

Rewriting logic is defined on top of an arbitrary equational logic. However, *membership equational logic* [BJM97] is its usual basis. It is defined by means of two classes of atomic sentences, *equations* $t = t'$ and sort *membership axioms* $t : s$, which identify terms and assign them a sort, respectively. These can be combined to produce Horn clauses of the form:

$$t = t' \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \qquad\qquad t : s \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j$$

Membership equational theories are pairs (Σ, E), and their models are Σ-algebras. Given a Σ-algebra A and an *assignment* $a : X \to A$ of variables in X to elements in A, a can be extended inductively to $\overline{a} : T_\Sigma(X) \to A$ like substitutions, with $\overline{a}(x) = a(x)$ and $\overline{a}(f(t_1, \dots, t_n)) = f^A(\overline{a}(t_1), \dots, \overline{a}(t_n))$. An atomic equation is satisfied $A, a \vDash u = v$ for A and a if $\overline{a}(u) = \overline{a}(v)$, and a membership axiom $A, a \vDash u : s$ if $\overline{a}(u) \in A_s$. Hence, a general sentence ϕ if $\bigwedge_i \phi_i$ is satisfied by A if for all assignments a such that $A, a \vDash \phi_i$ for all i, then $A, a \vDash \phi$. A (Σ, E)-algebra is a Σ-algebra that satisfies all the equations and axioms in E.

Ground terms T_Σ are purely syntactical elements, but they can be identified up to provable equality $=_E$ with the equations and axioms in E. The *initial term algebra* $T_{\Sigma/E} := T_\Sigma / =_E$ is the quotient of T_Σ modulo this equality relation. In other words, it is the initial object of the category of (Σ, E)-algebras whose morphisms are Σ-homomorphisms. Although its elements $[t]$ are equivalence classes, we will usually write simply t when this is well-defined.

Rewriting is a correct and complete proof method for this equality, with $t =_E t'$ iff $t \to_E^* t'$. A term t can be rewritten $t \to_E t'$ to another term t' by an equation $u = v$ or $v = u$ if there is a position p and a substitution σ such that $t|_p = \sigma(u)$ and $t' = t[p / \sigma(v)]$. For a general conditional equation to be applied, the conditions $\sigma(u_i) =_E \sigma(u_i')$ and $\sigma(v_j) : s_j$ must be satisfied too. While testing equality is in general an undecidable problem, known as the *word problem*, if equations can be oriented and transformed to an equivalent convergent rewriting system, equality can always be checked by reducing terms to their normal forms and comparing them syntactically.[6] Some very frequent structural equations like commutativity $f(x, y) = f(y, x)$, associativity $f(x, f(y, z)) = f(f(x, y), z)$, and identity $f(x, e) = x = f(e, x)$ cannot be oriented, and they are usually set aside $E \cup B$ so that terms are considered and equations are applied modulo B with ad hoc procedures.

A *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ is a membership equational logic (Σ, E) with a set R of rewrite rules. From a specification point of view, terms and equations describe the structure of the state, and rewrite rules represent changes or actions that modify this state. A possibly conditional rewrite rule has the form:

$$l \Rightarrow r \qquad \text{if } \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \wedge \bigwedge_k w_k \Rightarrow w_k'$$

Conditions of the third type are named *rewriting conditions* and are satisfied if the instance of each w_k can be rewritten in zero or more steps with any rules in R to match w_k'. These statements induce a transition relation $\to_R$ on T_Σ. Formally, a rule as above rewrites a term t to t', $t \to_R t'$, if there is a position p in t and a substitution σ such that $t|_p =_E \sigma(l)$ and $t' =_E t[p/\sigma(r)]$; and $\sigma(u_i) =_E \sigma(u_i')$ for all i, $\sigma(v_j) \in T_{\Sigma,s_j}$ for all j, and $\sigma(w_k) \to_R^* \sigma(w_k')$ for all k. Rules can be applied in any position within the term, so they can easily capture local and concurrent transformations.

Operationally, equations are interpreted as simplification rewrite rules, which should be confluent and terminating to make equality decidable. On the contrary, rewrite rules R in a rewrite theory need not be confluent or terminating, because they represent nondeterministic change. However, rules are applied on top of the equational specification, modulo the equations, so a certain coherence between them is convenient. We say that R is *coherent* with E if for all $t, t', u \in T_\Sigma(X)$ such that $t \to_R^1 t'$ and $t \to_E^! u$, there is a $u' \in T_\Sigma(X)$ such that $u \to_R^1 u'$ and $t' =_E u'$. Remember that terms are reduced to their normal forms to test equality, so coherence guarantees that no rewrite is lost when looking for them only at the canonical form. Hence, rules can be applied on a term with the relation $\to_R^1 := \to_E^! \circ\!\!\!\!\to \to_R$ that is well defined for every representative of a class in $T_{\Sigma/E}$, whenever the aforementioned conditions on equations E and rules R are satisfied.

Under the previous conditions, a rewrite theory can be seen as a labeled transition system $(T_{\Sigma/E}, \Lambda, \to_R^1)$ where the actions are the labels assigned to the rules. More generally, the true concurrent model of rewrite theories are *rewriting systems*, i.e. categories with the structure of a (Σ, E)-algebra, with a natural transformation for each rule, and where morphisms can be identified with equivalence classes of proof terms for the following deduction rules. The complete definition can be found in [Mes92].

1. *Reflexivity*. For each $t \in T_\Sigma(X)$,

$$\frac{}{t : t \to t}$$

2. *Equality*.

$$\frac{\alpha : u \to v \quad u =_E u' \quad v =_E v'}{\alpha : u' \to v'}$$

3. *Congruence*. For each $f : k_1 \cdots k_n \to k$ in Σ, and $t_1, t_1', \ldots, t_n, t_n' \in T_\Sigma(X)$,

$$\frac{\alpha_1 : t_1 \to t_1' \quad \cdots \quad \alpha_n : t_n \to t_n'}{f(\alpha_1, \ldots, \alpha_n) : f(t_1, \ldots, t_n) \to f(t_1', \ldots, t_n')}$$

[6] The Knuth-Bendix and other completion procedures are available to transform equations into a confluent rewriting system whenever possible. We specify them in Maude by using strategies in Section 6.5.

4. *Replacement*. For each rule $r : t \Rightarrow t'$ if $\bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \wedge \bigwedge_k w_k \Rightarrow w_k'$ in R with variables $\{x_1, \ldots, x_n\}$,

$$\frac{\begin{array}{cc} \alpha_1 : p_1 \to p_1' \quad \cdots \quad \alpha_n : p_n \to p_n' & \cdots \quad \sigma(u_i) =_E \sigma(u_i') \quad \cdots \\ \cdots \quad \beta_k : \sigma(w_k) \to \sigma(w_k') \quad \cdots & \cdots \quad \sigma(v_j) \in T_{\Sigma, s_j} \quad \cdots \end{array}}{r(\overline{\alpha}, \overline{\beta}) : \sigma(t) \to t'[x_1 / p_1', \ldots, x_n / p_n']}$$

with the substitution $\sigma : \{x_1, \ldots, x_n\} \to T_\Sigma(X)$ such that $\sigma(x_i) = p_i$.

5. *Transitivity*.

$$\frac{\alpha_1 : t_1 \to t_2 \quad \alpha_2 : t_2 \to t_3}{\alpha_1 ; \alpha_2 : t_1 \to t_3}$$

Two proof terms α_1 and α_2 are equivalent if they represent the same concurrent computation as described in [Mes92]. For example, the proof term $(\alpha_1 ; \alpha_2) ; \alpha_3$ is the same as $\alpha_1 ; (\alpha_2 ; \alpha_3)$ since the transitive axiom is associative. An initial model $\mathcal{T}_\mathcal{R}$ can be associated to a rewrite theory $\mathcal{R}$ by taking the initial object in the category of $\mathcal{R}$-rewriting systems related by appropriate morphisms. One-step rewrites are derivations using only the rules 2, 3, and 4, except for the derivation of the sequents of rewriting conditions. The *sequential* one-step relation $\to_R^1$ of the previous paragraph is a one-step rewrite where the replacement rule has only been applied once, except again in rewriting conditions. Otherwise, *parallel* one-step rewrites can replace multiple arguments of a symbol at once.[7]

Rewriting logic can be extended to real-time systems by adding time annotations to the rewrite rules [Ölv14], to probabilistic systems [AMS06], etc. Moreover, rewrite theories can be analyzed both at the ground and at the symbolic level, where problems like symbolic reachability $\exists \overline{x}\ t(\overline{x}) \to t'(\overline{x})$ can be expressed.

2.5 Maude

Maude [CDE+20, CDE+07] is a specification and programming language whose programs are exactly rewrite theories. Actually, rewriting logic was introduced along with this language from the very beginning [Mes92]. Its equational sublanguage and its syntax are very close to the OBJ language [GWM+00], and other languages of this family like ELAN [BKK+01] and CafeOBJ [DF02] also support rewriting logic. Maude is in active development, being currently extended with new features for symbolic reasoning via unification, narrowing, variants, and connections with SMT solvers; for the interaction with the outside world using external objects; and for controlling rewriting with the strategy language reported in this thesis [DEE+20].

Maude specifications are the straight translation to ASCII of the previous mathematical notation for equations, membership axioms, and rules. Specifications are organized in modules that can include or extend other modules. Some standard modules are provided in the Maude *prelude* specifying natural and floating-point numbers, lists, strings, sets, reflective operations, and so on. Pure equational theories are represented in *functional modules*, started by the **`fmod`** keyword and closed with **`endfm`**. For example, the following `MY-NAT-LIST` module includes in the builtin `NAT` module of natural numbers and defines lists of them.

```
fmod MY-NAT-LIST is
  protecting NAT .

  sorts List Pair .
  subsorts Nat Pair < List .
```

[7]When discussing membrane systems in Chapter 8, we will see that the evolution step of this computational model does not coincide with a sequential rewriting step, but with a parallel one.

```
  op nil : -> List [ctor] .
  op __  : List List -> List [ctor assoc id: nil] .

  vars N M : Nat .
  var  L : List .

  mb N M : Pair .

  op length : List -> Nat .
  eq length(nil) = 0 .
  eq length(N L) = length(L) + 1 .
endfm
```

Sorts are introduced with the **sort**(**s**) keyword, and subsort relations are given with **subsort**(**s**). Operators are declared with the **op** keyword followed by its name, signature and attributes. Underscores mark the gaps where the operator arguments should be filled in a general mixfix syntax, while prefix syntax is used for operators without underscores. Attributes declare syntactic properties of the operators like their precedence and formatting, metadata associated to them, and other more semantic properties like whether they are constructors (`ctor`) of their data types. Structural axioms like `assoc`iativity, `comm`utativity, and `id`entity that are handled specifically by the Maude engine are also specified as attributes. For example, the list concatenation operator `__` is declared associative and having the empty list `nil` as identity element. Membership axioms and equations are declared with the **mb** and **eq** keywords respectively, and they can be prepended a **c** to make conditional statements. For instance, we could have written `cmb N M : Pair if N < M /\ N > 10 .`. Each connected component of sorts via the subsort relation is identified with a kind of the order-sorted signature defined by the module. Maude implicitly completes each kind with a common supersort or *kind sort* that can be written $[s]$ for any sort s in the component. Formally, an operator declaration $f : s_1 \cdots s_n \to s$ can be understood as the declaration of both a kind-level operator $f : [s_1] \cdots [s_n] \to [s]$ and a membership axiom $f(x_1, \ldots, x_n) : s$ where $x_k : s_k$. For example, we could have written the membership axiom in the previous module as `op __ : Nat Nat -> Pair [ctor ditto] .`, where `ditto` copies the attributes of the first declaration of the same kind-level operator. Subsort declarations can be interpreted similarly as membership axioms. Terms whose least general sort is the kind sort $[s]$ are usually error terms, like those produced by partial functions when called outside their domain. These functions should be declared with a curly arrow $\rightsquigarrow$ instead of a normal one $\to$, so that no statement is made about the sort membership of the result.

The Maude execution engine sees equations as rightward-oriented simplification rules, and applies them exhaustively to obtain canonical forms of the terms modulo structural axioms. For this procedure to be sound, the simplification relation $\to_E$ should be confluent and terminating modulo the structural axioms. Modules satisfying these requirements are called *admissible* modules, but Maude does not check this automatically, since it is generally undecidable. The Maude Formal Environment [DRÁ11] includes tools for checking these properties. Canonical forms can be obtained with the `reduce` or `red` command of the Maude interpreter.

```
Maude> reduce 1 2 .
rewrites: 1
result Pair: 1 2
Maude> reduce length(3 2 1 0) .
rewrites: 11
result NzNat: 4
```

Equations may not be applied in the order they appear in the module, but if the `owise` attribute is added to an equation, it will always be used after all equations without this attribute have failed. Conditional equations are introduced with the **ceq** keyword instead of **eq** and their conditions follow the syntax of the formal equations explained in the previous section. In addition to the standard equality and sort membership clauses, matching conditions $t := t'$ allow free variables in their lefthand sides to be instantiated by matching so that both sides coincide. Variable values are propagated from the pattern to the condition fragments, and among them from left to right.

System modules specify rewrite theories by adding rules, and they are delimited by the **mod** and **endm** keywords.

```
mod NAT-LIST-MEAN is
  protecting MY-NAT-LIST .

  vars N M  : Nat .
  vars L L' : List .

  rl  [mean]  : N M => (N + M) quo 2 .
  crl [other] : L => L' if N := length(L) /\ N = 10 /\ L N => L' M .
endm
```

This system module NAT-LIST-MEAN imports the functional module MY-NAT-LIST and adds two rewrite rules introduced by the **rl** and **crl** keywords. They are given the labels mean and other between brackets, which can also be declared with the label attribute. System modules may also contain the statements of functional modules, and they may import functional or system modules with the protecting, extending, and including keywords. The difference between these importation modes is whether the new module adds *junk* or *confusion* to the elements of the imported modules, and this is not checked by Maude. In summary, protecting M means that neither new terms are added to the sorts of the imported module, nor they are identified with new equations, and that no more rewriting paths are defined for them; extending M may add new terms and rewriting paths to the sorts of M; and including may also identify terms. Conditions in rewrite rules are written and evaluated like equational conditions, but they may also include rewriting fragments like `L N => L' M` above. These are satisfied if a term matching the righthand side pattern can be obtained from the initial one by rewriting with the rules in the module. Variables in these patterns can be used in the following condition fragments and in the righthand side of the rule. Rules can be executed with the rewrite or rew command in the interpreter.

```
Maude> rew 7 1 5 9 .
rewrites: 58
result NzNat: 6
Maude> rew [2] 7 1 5 9 .
rewrites: 28
result Pair: 4 9
```

The number of rewrites can be bounded by a number within brackets as above, and an alternative rewriting command frewrite or frew uses a fair strategy to decide which and where rules are applied. Moreover, the search command can be used to search terms reachable by rewriting from a given one and inspect the rewriting paths that lead to them.

```
Maude> search 7 1 5 9 =>* N s.t. N =/= 6  .

Solution 1 (state 10)
states: 11
N --> 5

Solution 2 (state 11)
states: 12
N --> 7
```

$$\begin{tikzcd}
\texttt{TH1} \arrow[rr, "\texttt{View1}"] & & \texttt{M1} \\
\vdots & & \vdots \\
\texttt{THn} \arrow[rr, "\texttt{Viewn}"] \arrow[d] & & \texttt{Mn} \arrow[d] \\
\texttt{PM\{X1 :: TH1, …, Xn :: THn\}} \arrow[rr] & & \texttt{PM\{View1, …, Viewn\}}
\end{tikzcd}$$

Figure 2.3: Diagram of a parameterized module instantiation.

```
No more solutions.
states: 12
```

The complete rewriting path can be seen with the `show path` command:

```
Maude> show path 10 .
state 0, List: 7 1 5 9
===[ rl N M => (N + M) quo 2 [label mean] . ]===>
state 1, List: 4 5 9
===[ rl N M => (N + M) quo 2 [label mean] . ]===>
state 5, Pair: 4 7
===[ rl N M => (N + M) quo 2 [label mean] . ]===>
state 10, NzNat: 5
```

For equations and rules to be executable, every variable in the righthand side and every free variable in the condition must be bound in the lefthand side or a previous condition fragment. However, these statements can be annotated with the `nonexec` attribute if they are not intended to be executed. In the case of rules, this is useful for narrowing or when they are explicitly applied with a partial substitution at the metalevel or using the strategy language. Admissible system modules are those whose equational part is admissible and whose rule application relation $\rightarrow^1_R$ is coherent with equations, as explained in Section 2.4.

There are commands available in the Maude interpreter like `match` to get the matches of a term into another term, `unify` to get unifiers of two terms, and others related to narrowing, SMT solving, and to the strategy language. The latter are described in Chapter 3. More details about the language, its interpreter, related tools, and applications can be found in the Maude manual [CDE+20].

2.5.1 Parameterization

Maude supports *parameterized programming* [Gog84, CDE+20; §6.3] in its modules. Parameterization has three basic building blocks: theories, parameterized modules, and views. *Theories* describe the interface of parameterized modules, by specifying the requirements that any actual parameter of those must satisfy, with a syntax almost identical to modules. They are delimited by the keywords **`fth`** and **`endfth`** for functional theories, and **`th`** and **`endth`** for system theories. However, their declarations are understood as formal objects, and the executability requirements that a module must obey are not demanded for theories. The simplest one, although extensively used, is the following theory `TRIV` specifying a single parameter sort:

```
fth TRIV is
  sort Elt .
endfth
```

A *parameterized module* includes in its header `PM{X1 :: TH1, …, Xn :: THn}` a list of one or more formal parameters, each bound to a theory and identified by a name. In its body, the parameter sorts are referred to by their formal names prefixed by the parameter name and a dollar sign `$`, while the formal operators are cited by their original names unchanged. For instance, the following parameterized module `LIST` defines a list of elements of its parameter sort.

```
fmod LIST{X :: TRIV} is
  protecting NAT .

  sort List{X} .
  subsort X$Elt < List{X} .

  op nil : -> List{X} [ctor] .
  op __ : List{X} List{X} -> List{X} [ctor comm assoc id: nil] .

  var E : X$Elt .
  var L : List{X} .

  op length : List{X} -> Nat .
  eq length(nil) = 0 .
  eq length(E L) = length(L) + 1 .
endfm
```

How a module adheres to a theory is specified using a *view*, which maps the formal objects in the theory to the actual objects in the chosen target module where the theory is interpreted. Views are then used to instantiate the parameterized modules as depicted in Figure 2.3. For example, the following view interprets `NAT` as a `TRIV` by mapping the formal sort `Elt` to the sort `Nat` of natural numbers.

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
```

Then, the parameterized module `LIST{X :: TRIV}` is instantiated by the view `Nat` to produce the module `LIST{Nat}` of lists of natural numbers. Such module instantiation expressions can be used in importation statements.

```
fmod COUNTDOWN is
  protecting LIST{Nat} .
  op countdown : -> List{Nat} .
  eq countdown = 5 4 3 2 1 0 .
endm
```

Views may map multiple sorts and operators, include other theories in `including` mode, and import modules that are used by their formal objects. Moreover, they can also be parameterized as in the following `List` view, so that lists of lists of natural numbers can be instantiated with `LIST{List{Nat}}`.

```
view List{X :: TRIV} from TRIV to LIST{X} is
  sort Elt to List{X} .
endv
```

2.5.2 Reflection and internal strategies

Rewriting logic is a reflective logic, whose objects and operations can be consistently represented in itself. Maude offers a predefined *universal theory* [CDE+20; §17] to metatheoretically represent terms, equations, rules, modules, and so on. Operations like matching, reduction, and rule application can be programmed generically using regular operators and equations, but Maude provides special operators backed by the object-level implementation in C++ to allow efficient reflective computations. Metarepresentations can in turn be metarepresented and terms be moved between different levels, thus yielding arbitrarily high reflective towers, if required.

This universal theory is specified in `META-LEVEL` and its imported modules, and it relies on the `Qid` sort of *quoted identifiers*, arbitrary words prefixed by an apostrophe. A variable `X` of sort `Nat` is metarepresented as the quoted identifier `'X:Nat`, and the constant `'Nat` of sort `Qid` is `''Nat.Qid`. Terms with arguments are represented using the operator `op _[_] : Qid NeTermList -> Term` like `'_+_['X:Nat, 's_['0.Zero]]` for `X + s 0`. Operator declarations, equations, rules, and similar statements are represented as terms with a syntax similar to the object-level reference. For example, the operator `+` may have a declaration `op '_+_ : 'Nat 'Nat -> 'Nat [comm assoc] .` and be involved in an equation `eq '_+_['X:Nat, '0.Zero] = 'X:Nat [none] .` where the trailing brackets enclose the set of operator or statement attributes. Metamodules are terms with argument slots like `fmod_is_sorts_.____endfm` for each kind of module component. Auxiliary functions `getOps`, `getEqs`, `getRls`, etc., are defined to obtain these components.

Operations are accessible through *descent functions* like `metaMatch` for matching, `metaApply` and `metaXapply` for rule application, `metaReduce` for equational reduction, `metaRewrite` to rewrite a term as in the `rewrite` command, `metaParse` for parsing terms in a module, etc.

```
op metaReduce  : Module Term ⇝ ResultPair [special (...)] .
op metaRewrite : Module Term Bound ⇝ ResultPair [special (...)] .
op metaApply   : Module Term Qid Substitution Nat ⇝ ResultTriple? [...] .
op metaXapply  : Module Term Qid Substitution Nat Bound Nat
                   ⇝ Result4Tuple? [...] .
op metaMatch   : Module Term Term Condition Nat ⇝ Substitution? [...] .
op metaXmatch  : Module Term Term Condition Nat Bound Nat ⇝ MatchPair? [...] .
op metaParse   : Module VariableSet QidList Type? ⇝ ResultPair? [...] .
```

They all include the `special` attribute, which means that they are directly implemented by the interpreter in C++. For instance, `metaReduce` receives the metarepresentations of a module and a term, and produces a pair containing the reduced term and its calculated sort. Other predefined functions allow obtaining the metarepresentation of a term (`upTerm`) or the object-level term from its metarepresentation (`downTerm`). The metarepresentation of loaded modules can be obtained with `upModule` given the module name and a Boolean flag indicating whether a flat version (with all imports resolved) of the module is required, like in `upModule('NAT, false)`. The complete specification of the metalevel is in the Maude prelude and explained in [CDE+20; §17].

One of the applications of reflection is writing tools to analyze Maude specifications in Maude itself, like those gathered in the Maude Formal Environment [DRÁ11] and the interactive theorem prover [CPR06]. Moreover, reflection can also be used to extend the language with new features and commands, and to program interactive interfaces in Maude. The clearest example is Full Maude [CDE+20; Part II], an extended Maude interpreter written in Maude with all the features of the standard Maude interpreter and some others like object-oriented modules and tuple types. Many features that are currently available in Core Maude, as the C++-implemented interpreter is usually called, were first experimented in Full Maude, including the strategy language this thesis is about. Other Maude extensions have been programmed using the reflective features and extending Full Maude, like Real-Time Maude [Ölv14] for real-time rewriting specifications.

Strategies can also be specified using the reflective features of Maude. Metalevel programs have been used as internal strategies since the beginning of Maude [CM96, CM97]. For example, the following functional module specifies a strategy that reduces lists in `NAT-LIST-MEAN` by applying the `mean` rule from left to right.

```
fmod NAT-LIST-STRAT is
  protecting META-LEVEL .

  op mean-l2r : Term → Term .

  vars T1 T2 T : Term .
  var  TL : TermList .

  eq mean-l2r('__[T1, T2, TL]) = mean-l2r(getTerm(
    metaXapply(upModule('NAT-LIST-MEAN, false),
               '__[T1, T2, TL], 'mean, 'N:Nat <- T1 ; 'M:Nat <- T2,
               0, unbounded, 0))) .
  eq mean-l2r(T) = T [owise] .
endfm
```

The essential element in any internal strategy to control rewriting is the `metaXapply` operation with which rules are applied. In this case, it applies the rule `mean` on the term given as argument with a fixed value for the rule variables that targets its application to the first two elements. The other arguments delimit the depth where the rule can be applied, and the last one is an index to enumerate the multiple possible solutions until a `failure` result is obtained. If the list does not contain at least two elements, the pattern of the first equation fails and the second equation does not modify it. We can execute the strategy and obtain the result with the `reduce` command:

```
Maude> red mean-l2r('__['s_^7['0.Zero], 's_['0.Zero],
                        's_^5['0.Zero], 's_^9['0.Zero]]) .
rewrites: 24
result GroundTerm: 's_^6['0.Zero]
```

Its argument is the metarepresentation of `7 1 5 9`, since natural numbers are represented in Peano notation and iterated operators can be collapsed with the `^`*n* syntax. Although this strategy is simple, its metalevel program is verbose and using it requires writing and reading terms at the metalevel. More complex control mechanisms and applying rules that may fail or produce multiple solutions complicate the understanding of internal strategies, so this is why the object-level strategy language detailed in Chapter 3 was proposed.

2.5.3 External objects

Maude allows connecting with the external world by means of some *external objects* for sockets, standard text streams, files, external processes, and metainterpreters. They are called objects because messages can be sent to and received from them using the object-oriented facilities of the `CONFIGURATION` module of the Maude prelude. This module declares a multiset or soup that can be filled with custom messages and objects.

```
sorts Oid Cid Object Msg Portal Configuration .
subsort Object Msg Portal < Configuration .
op <_:_|_> : Oid Cid AttributeSet → Object [ctor object] .
op none : → Configuration [ctor] .
op __ : Configuration Configuration
          → Configuration [ctor config assoc comm id: none] .
op <> : → Portal [ctor] .
```

Objects are usually collections of attributes of the form *key*: *value* identified by constants of sort `Oid` and associated to classes of sort `Cid`. Messages are operators defined with the `msg` attribute whose first and second arguments are usually the object identifiers of its receiver and sender. User-defined objects read and introduce messages in the soup by means of rewrite rules. While external objects are not present in the configuration (if not through the portal `<>`, which may be present when they are used), they introduce and consume messages by implicit rewrites. The command `erewrite` or `erew` conducts rewriting of these configurations following an object-fair strategy and handling external-object communication.

In this thesis, we will only use the standard streams to write interactive interfaces[8] and file objects to read programs from file, in the examples of Chapters 7 and 8. The `STD-STREAM` and `FILE` modules in the `file.maude` file of the Maude distribution declare the `stdin`, `stdout`, `fileManager`, and `file(`*n*`)` objects, and the messages `getLine`, `read`, `write`, `openFile`, `close`, etc., to communicate with them.

2.5.4 Model checking

As we have argued in Section 2.4, rewriting logic specifications can be seen as transition systems whose states are terms modulo equations and structural axioms, and whose transitions are sequential one-step rule rewrites. Temporal properties can be checked on this system using the Maude LTL model checker [EMS04], which is an integral part of Maude since its 2.0 version. It has been given many applications since then [Rie18, FCM+04, CFP+18, Oga17, CM17; ...].

The procedure to prepare a specification to be model checked is simple [CDE+20; §12] and relies on a few predefined modules available in the `model-checker.maude` file included in the Maude distribution. Atomic propositions are declared as regular Maude symbols and defined equationally using a predicate $\_ \models \_$ provided by the predefined `SATISFACTION` module.

```
fmod SATISFACTION is
  protecting BOOL .
  sorts State Prop .
  op _⊨_ : State Prop → Bool [frozen] .
endfm
```

The model checker is accessed through a special operator declared in the `MODEL-CHECKER` module, which returns either a Boolean value `true` if the property is satisfied, or a counterexample if it is not.

```
op modelCheck : State Formula ⇝ ModelCheckResult [special (...)] .
```

Formulae of sort `Formula` follow the syntax of the predefined `LTL` module, which is similar to the mathematical notation we have used here. There is also the possibility that the model checker does not terminate if the state space is large enough or the specification does not satisfy the decidability conditions mentioned in [CDE+20; §12]. More details are given when we introduce our extension of this model checker for strategy-controlled systems in Section 4.3.

At the algorithmic level, the builtin model checker is an optimized implementation of the standard on-the-fly explicit-state LTL approach described in Section 2.3.2. The input formula is negated and translated to a Büchi automaton using the LTL2BA algorithm [GO01] with some optimizations [SB00], and the model automaton is calculated on demand from the given initial state.

2.6 Strategy logics

In games and multiagent systems, many natural questions and properties talk about strategies. Does the starting player always have a strategy to win no matter what the other players do? Can a group of agents coordinate to maintain a given property? While strategies are usually fixed as part of the model specification in this thesis, strategic logics extend the usual temporal logics to allow *reasoning* about strategies in their formulae with existential and universal quantifiers on them. Relevant examples are Alternating-Time Temporal Logic [AHK02], the Strategy Logic

[8] In previous versions, Maude offered a different input/output facility called `LOOP-MODE`, which is currently deprecated in favor of this more flexible method.

of Chatterjee, Henzinger, and Pitterman [CHP10], the different Strategy Logic of Mogavero, Murano, Perelli, and Vardi [MMP⁺14], and their several variations. Model checking for these and related logics is an active research topic.

In this section, we will briefly describe the $\mathsf{SL}^{\prec}$ strategy logic [GMM20] as an example. Although it differs in various aspects from the most usual strategy logics, it is somehow more general and more convenient for finding a relation with the satisfaction problem associated to our model-checking problem in Section 4.6. The models of this logic are concurrent game structures, which can be understood as Kripke structures whose transitions are the result of the simultaneous decisions of each agent participating in the game.

Definition 2.6. *A* concurrent game structure *(CGS) is a tuple* $\mathcal{G} = (\mathrm{Ag}, \mathrm{Ac}, V, E, AP, \ell, v_0)$ *where* Ag *is a finite non-empty set of agents or players,* Ac *is a finite non-empty set of actions,* V *is a finite non-empty set of positions,* $E : V \times (\mathrm{Ag} \to \mathrm{Ac}) \to V$ *is a transition function,* AP *is a finite non-empty set of atomic propositions,* $\ell : V \to \mathcal{P}(AP)$ *is a labeling function, and* v_0 *is an initial position.*

In the definition, functions $\mathrm{Ag} \to \mathrm{Ac}$ represent the global decision of the agents and determine the next position of the game. Strategies are individual to each agent, and they are formalized as intensional strategies $\lambda : V^+ \to \mathcal{P}(\mathrm{Ac})$ that choose a subset of possible actions. We will use Strat to name that set of functions. The syntax of $\mathsf{SL}^{\prec}$ extends that of CTL* with an existential operator $\exists x\ \varphi$ that claims the existence of a strategy such that φ holds, a strategy binding operator $(a, x)\,\varphi$ that assigns a previously quantified strategy x to an agent a, and $x \preceq y$ that asserts that the strategy x is more restrictive than y.

$$\begin{aligned} \varphi &::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\,\varphi \mid (a,x)\,\varphi \mid x \preceq y \mid \mathbf{E}\,\psi \\ \psi &::= \varphi \mid \neg\,\psi \mid \psi \vee \psi \mid \bigcirc\,\psi \mid \psi\,\mathbf{U}\,\psi \end{aligned}$$

The existential quantifier $\exists x$ can also be seen as $\langle\!\langle x \rangle\!\rangle$. Other propositional operators and the universal strategy and path quantifiers are derived by the usual equivalences. We will not reproduce the complete semantics of the logic but concentrate on the strategic part. A formula φ is satisfied in a CGS $\mathcal{G}$ after having executed ρ as the following relation specifies:

$$\begin{aligned} \mathcal{G}, \chi, \rho \models \exists x\,\varphi &\iff \exists \lambda \in \mathrm{Strat} \quad \mathcal{G}, \chi[x \mapsto \lambda], \rho \models \varphi \\ \mathcal{G}, \chi, \rho \models (a,x)\,\varphi &\iff \mathcal{G}, \chi[a \mapsto \chi(x)], \rho \models \varphi \\ \mathcal{G}, \chi, \rho \models \mathbf{E}\,\psi &\iff \exists \pi \in \mathrm{out}(\chi,\rho) \quad \mathcal{G}, \chi, \pi, |\pi| - 1 \models \psi \\ \mathcal{G}, \chi, \rho \models x \preceq y &\iff \forall \pi \in V^* \quad \chi(x)(\rho\pi) \subseteq \chi(y)(\rho\pi) \end{aligned}$$

The partial function $\chi : \mathrm{Var} \cup \mathrm{Ag} \rightsquigarrow \mathrm{Strat}$ is an assignment of strategies to both variables and agents, updated and used by the semantic definitions above. $\mathrm{out}(\chi, \rho) \subset V^+$ is the set of all plays from ρ whose steps are acceptable by the strategies, i.e. such that $\pi_{n+1} = E(\pi_n, d)$ for a decision d such that $d(a) \in \chi(a)(\pi^{\leq n})$ for every agent $a \in \mathrm{Ag}$ in which $\chi(a)$ is defined. The path quantifier $\mathbf{E}$ of CTL* selects one of these allowed plays π and evaluates the LTL-like path formula ψ on π^k with the relation $\mathcal{G}, \chi, \pi, k \models \psi$ until a new state subformula is encountered.

Changing the underlying notion of strategy yields a different logic whose properties have a completely different meaning [JM14, BJ14]. Most variants of strategy logic consider deterministic strategies $\lambda : V^+ \to \mathrm{Ac}$ instead of nondeterministic ones, and some even restrict to positional strategies $\lambda : V \to \mathrm{Ac}$ that only depend on the last position of the game. The main strategy logic [MMP⁺14] imposes the additional restriction that all agents must be bound to a strategy, so that strategies completely determine the evolution of the game, $\mathrm{out}(\chi, \rho)$ denotes a single play, and operators like $\preceq$ and $\mathbf{E}$ become meaningless. Other variations are related to the meaning of strategy bindings (a, x), which may refine or replace the strategy assigned to the mentioned agent or the other agents. In this logic, strategies are replaced, but refinement can be stated with the $x \preceq y$ formula. The logic USL [CBC15] also targets nondeterministic strategies with operators that explicitly refine and revoke previous bindings.

The model-checking problem for $\mathsf{SL}^{\prec}$ (and similarly for other strategy logics) is deciding whether the statement $\mathcal{G}, \varnothing, \varepsilon \models \varphi$ holds for a formula φ without free variables. Its complexity lies in the $(k+1)$-EXPTIME-complete class, where k is the *simulation depth* of the formula φ, which is related to the alternation of $\exists$ and $\mathbf{E}$ quantifiers and negations. However, there are some usable model checkers for some restricted subsets of strategy logics [CLM15, CLM+14], which are even able to synthesize those strategies whose existence is claimed in the formula.

As we will discuss in Sections 4.6 and 4.8, our model-checking problem cannot be reduced to those of the strategy logics, but its associated satisfaction problem can be seen as a very particular case of $\mathsf{SL}^{\prec}$ model checking.

2.7 Probabilities and quantitative verification

In the previous sections and in most of this thesis, we deal with qualitative and absolute properties of nondeterministic models. However, quantitative properties such as cost, time, and probabilities are also interesting and relevant for many applications. Moreover, whereas the unquantified nondeterminism of standard transition systems equalizes all possible behaviors regardless of their likeliness, assigning probabilities to their transitions allows obtaining less strict results that may still be useful in practice. We start this section by introducing a few basic notions about probabilities so that we can define generalizations of the abstract transition systems presented in Section 2.2 where probabilistic behavior can be modeled. Model checking classical and intrinsically probabilistic temporal properties is then discussed.

Probabilities are numerical descriptions of how likely events are to occur, conventionally represented as real numbers from 0 for an almost impossible situation to 1 for an almost sure one. Given the set of possible outcomes of an experiment X, events are formalized as subsets $A \subseteq X$ so that the logical connectors used to describe them are translated to complements, unions and, intersections. When the set X is countable and the probability $P : X \to [0,1]$ of every potential result is known, the probability of an event A can be calculated as the sum or series $\sum_{x \in A} P(x)$ of their individual probabilities. The probability of the event X and so the sum of all values of P must be 1, since these are all possible outcomes of the experiment. This convenient description of probabilities $P : X \to [0,1]$ is sometimes called *probability distribution*, and we denote by $\mathrm{Dist}(X)$ the set of all probability distributions on X. Moreover, for an assignment $f : X \to V$ of values in a vector space V to the elements of X, the *expected value* of f is defined as the weighted mean $\sum_{x \in X} P(x) f(x)$. For example, the possible outcomes of throwing a fair dice are $X = \{1, \ldots, 6\}$, and the probability of obtaining any of these values is $P(n) = 1/6$. Hence, the event of getting an odd dice roll is $\{1, 3, 5\}$ and its probability is $1/2$. If we move in a board as many positions as indicated by the dice, the expected value of the number of moves is then $1 \cdot 1/6 + \cdots + 6 \cdot 1/6 = 7/2$.

While combinatorics and the previous description of probabilities are enough for most of the problems we will find here, they are insufficient when uncountable spaces are considered. And they are indeed considered, since time takes values in the set of positive real numbers and executions are represented as sets of infinite words, and none of these are countable. In these cases, it is perfectly reasonable that all individual outcomes of an event with positive probability have probability zero, and also that some subsets cannot be soundly assigned a probability. The general framework of measure theory should be used to properly handle them [Kal21]. A *σ-algebra* $\Sigma \subseteq \mathcal{P}(X)$ on a set X is a non-empty collection of subsets of X that is closed under complement and countable unions. For example, the set $\mathcal{P}(X)$ of all subsets of X is always a σ-algebra and it is called the discrete one. Another relevant example is the Borel σ-algebra $\mathcal{B}(X)$ of a topological space X, defined as the smallest one containing all open sets in X. A *measure* $\mu : \Sigma \to [0,\infty]$ is a σ-additive function such that $\mu(\emptyset) = 0$, where σ-additive means that $\mu(\bigcup_{k=1}^{\infty} A_k) = \sum_{k=1}^{\infty} \mu(A_k)$ for all countable collections $(A_k)_{k=1}^{\infty}$ of disjoint sets in Σ. Given a set X and a σ-algebra Σ on X, (X, Σ) is called a *measurable space* and a function $f : X \to Y$ to another measurable space (Y, Σ') is *measurable* if $f^{-1}(B) \in \Sigma$ for all $B \in \Sigma'$. Measurable functions induce measures μ' on their images by $\mu'(B) := \mu(f^{-1}(B))$. A measurable space (X, Σ) along with a measure $\mu : \Sigma \to [0,\infty]$ is called a *measure space* (X, Σ, μ).

A *probability measure* is a measure μ such that $\mu(X) = 1$, and (X, Σ, μ) is then called a *probability space*. Measurable functions between probability spaces, and specially those whose image is in $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$, are called *random variables*. The *expected value* of a random variable $\mathbb{E}[f]$ is intuitively the weighted average of the values of f, as we have previously defined it. Formally, it is defined as its integral $\int_X f \, d\mu$ with respect to the measure, which coincides with a sum or series $\sum_{x \in X} f(x)$ in the countable case. For example, the discrete σ-algebra can be considered in the set $X = \{1, \dots, 6\}$ of all possible outcomes of a die, a measure can be defined by σ-additivity from $\mu(\{n\}) = 1/6$ as before, and the expected value $\mathbb{E}[\mathrm{id}]$ of the identity $\mathrm{id} : D \to \mathbb{R}$ is again 7/2. In the countable context, measures $\mu(B) = \sum_{x \in B} P(x)$ and distributions $P(x) = \mu(\{x\})$ will be deliberately confused and used interchangeably. We denote by $\mathrm{Mes}(X, \Sigma)$ or simply $\mathrm{Mes}(X)$ the set of all probability measures on (X, Σ).

Finally, an indexed family $(f_n)_{n \in I}$ of random variables with $f_n : X \to Y$ between two probability spaces X and Y is called a *random process*. When $I = \mathbb{N}$ or $I = \mathbb{R}$ we say that the random process is a discrete-time or a continuous-time one respectively. While the standard definition above refers to random variables, we prefer to see them as indexed families of probability measures or distributions on Y.

2.7.1 Specification of probabilistic systems

Abstract probabilistic systems are usually modeled using a probabilistic enrichment of the transition systems introduced in Section 2.2 [CHV+18, BK08; §10]. The most basic extension consists of assigning probabilities to the successors of each state.

Definition 2.7. *Given a set S of states and an initial probability distribution P_0 on S, a* discrete-time Markov chain *(DTMC) is (S, P, P_0) where $P : S \times S \to [0, 1]$ satisfies $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$.*

In other words, P fixes a probability distribution $P_s(s') = P(s, s')$ on the successors of every state $s \in S$. Executing a discrete-time Markov chain consists in visiting its states according to the probabilities of their transitions. An execution π can be conceived as

- a discrete-time random process on the states, whose initial distribution is $\pi_0 = P_0$ and such that $\pi_{n+1}(s) = \sum_{r \in S} P(r, s) \, \pi_n(r)$, or alternatively
- an element of a probabilistic space (S^ω, Σ, μ) on the executions of the underlying transition system. In effect, the probability of executing a finite sequence of steps $w = s_0 \cdots s_n$ is
$$\hat{P}(w) = \pi_0(s_0) \, P(s_0, s_1) \cdots P(s_{n-1}, s_n),$$
so the joint probability of all executions in $\{w\}S^\omega$ is $\hat{P}(w)$ too. Taking the Borel σ-algebra engendered by the topology of the infinite words space described in Section 2.1, whose generators are precisely the cylinder sets $\{w\}S^\omega$, there exists a probability measure μ such that $\mu(\{w\}S^\omega) = \hat{P}(w)$, extended by σ-additivity to the whole space.

In this context, it is possible to obtain sound statistical and quantitative results about the system behavior. However, discrete-time Markov chains are not always convenient to model computational systems, where nondeterministic behavior is usually mixed with probabilistic one. This problem is addressed by the following structures in a different but essentially equivalent way:

Definition 2.8. *Given a set S of states and an initial probability distribution P_0 on S,*

1. *A* Markov decision process *(MDP) is (S, A, P, P_0) where A is a set of actions and $P : S \times A \times S \to [0, 1]$ a family of probability distributions satisfying $\sum_{s' \in S} P(s, a, s') = 1$ for all $s \in S$ and $a \in A$.*
2. *A* probabilistic abstract reduction system *(PARS) [BG05] is (S, R) where $R \subseteq S \times \mathrm{Dist}(S)$.*

Intuitively, Markov decision processes combine the nondeterministic choice of an action with the quantified selection of a successor for each action. The former may be seen as controllable by the user, while the latter are uncontrollable and produced by the environment. In probabilistic abstract reduction systems, a probability distribution is chosen among those defined for a state without explicit reference to actions. MDP and PARS are almost equivalent, since the relation R of a PARS can be defined as $\{(s, P_{s,a}) : s \in S, a \in A\}$ with $P_{s,a}(s') = P(s, a, s')$ for any given MDP, and the converse is also possible with the appropriate set of actions. Hence, in the following, we will only deal with Markov decision processes. In both cases, like in discrete-time Markov chains, the selection of the next step is governed by memoryless probabilities that only depend on the current state.[9]

Numerical results about the previous systems may drastically depend on how the nondeterministic choices are resolved. Strategies play an important role in this setting in order to quantify this nondeterminism and allow a complete statistical analysis. Also known as *schedulers* and *policies*, they are defined like the intensional strategies of Section 2.2, as functions $\lambda : S^+ \to \mathrm{Dist}(A)$ that assign to every partial execution a probability distribution on the next available actions. A Markov decision process $D = (S, A, P, P_0)$ along with a strategy λ induce a discrete-time Markov chain $\lambda(M) := (S^+, P', P_0)$ based on its execution tree with

$$P'(ws, wss') := \sum_{a \in A} \lambda(ws)(\{a\}) \cdot P(s, a, s')$$

Hence, quantitative properties of Markov decision processes can be enunciated as the maxima or minima among all possible instantiations.

We conclude with another formalism to describe probabilistic systems, the continuous-time version of Markov chains, used to describe biological, chemical, and telecommunication processes among others.

Definition 2.9. *Given a set S of states and an initial probability distribution P_0 on S, a* continuous-time Markov chain *(CTMC) is (S, R, P_0) where $R : S \times S \to [0, \infty)$ is a transition rate matrix.*

The transition rate $R(s, s')$ is the parameter of an exponential probability distribution describing how fast the transition $s \to s'$ is expected to occur.[10] When executing a continuous-time Markov chain, all successors of a state enter a race condition and the first transition to be triggered yields the next state of the sequence. Forgetting about measuring the time spent in each state, a CTMC can be embedded into a DTMC by taking $P(s, s') = R(s, s') / \sum_{s'' \in S} R(s, s'')$ unless this denominator is zero, where $P(s, s') = 1$ iff $s = s'$.

Specializations for rewriting systems. Specialization of these constructs for rewriting systems have also been considered in the literature [ALY20, AMS06], exploiting the additional structure of terms and rewrite rules. In Meseguer, Aga, and Sen [AMS06], *probabilistic rewrite theories* $\mathcal{R} = (\Sigma, E, R, \pi)$ are defined as rewrite theories where rules $r : t(\overline{x}) \Rightarrow t'(\overline{x}, \overline{y})$ if $C(\overline{x})$ include additional variables $\overline{y}$ whose values are given by a function π_r mapping matching substitutions on $\overline{x}$ to probability measures on the substitutions for $\overline{y}$. Formally, let $\mathrm{CGS}(\overline{x})$ be the set of canonical ground substitutions on the set of variables $\overline{x}$, and $[\![C]\!]$ the subset of those that satisfy the condition C. Given a substitution $\sigma \in [\![C]\!]$, the functions $\pi_r : [\![C]\!] \to \mathrm{Mes}(\mathrm{CGS}(\overline{y}))$ turn the set of canonical ground substitutions for $\overline{y}$ into a measurable space for an appropriate σ-algebra. A rewriting step still involves a nondeterministic choice of the rewrite rule, position, and matching substitution σ where it is applied, but then the probability measure $\pi_r(\sigma)$ determines the value of the $\overline{y}$ variables and so the rewritten term. An extension of Maude called PMAUDE [AMS06] allows specifying this kind of rules with the syntax

[9]Markov chains and decision processes are named after the Russian mathematician Andrey Markov, who studied this kind of memoryless random processes.

[10]An exponential distribution of parameter λ is a distribution that assigns probability $1 - e^{-\lambda x}$ to the intervals $[0, x]$. It is said to be memoryless, since the probability of $[s + t, \infty)$ divided by that of $[s, \infty)$ coincides with that of $[t, \infty)$.

```
crl t(x̄) ⇒ t′(x̄, ȳ) if C(x̄) with probability ȳ := π(x̄) .
```

where π is chosen from an extensible collection of standard probability distributions. PMAUDE modules can be statistically simulated with the VESTA tool and its continuations like MULTI-VESTA [SV13a]. However, results are only meaningful when the nondeterminism is completely removed, as we have mentioned before. An actor-based framework called Actor PMAUDE is provided for this purpose.

Another simpler approach to include probabilities in rewriting-based languages is incorporating them through strategies, as done by ELAN [BK02] and by Porgy [FKP19] (see Section 2.2.3). These strategy languages include an operator that receives a list of strategies along with their probabilities and chooses one of them accordingly. This operator is written `PC(`α_1 `:` p_1`, ...,` α_n `:` p_n`)` in an extension of ELAN and `ppick(`α_1`, ...,` α_n`, {`p_1`, ...,` p_n`})` in Porgy, where probabilities can also be calculated dynamically using a Python script. On the contrary, the probabilistic strategy language implemented in the PSMaude tool [BÖ13] removes the nondeterminism by specifying the probabilities for the choice of rules, contexts, and substitutions separately and with a dedicated syntax for each case.

2.7.2 Model checking probabilistic systems

While the verification questions in Section 2.3 always expect a Boolean answer, the analysis of probabilistic models can also yield quantitative results. We can now calculate the probability of entering an undesired state, the expected number of steps until a protocol converges, etc. Given a reward or cost function $f : S \to \mathbb{R}^+$ on the states, we can also calculate the expected values of the total reward $\sum_{n=0}^{\infty} f(\pi_n)$, the instantaneous reward $f(\pi_k)$ at a given step k, the average or steady-state reward $\lim_{n\to\infty} \sum_{k=1}^{n} f(\pi_k) / n$, and other variations.

Model checking does also make sense in the probabilistic setting, either against classical or probabilistic temporal logics. Like in Kripke structures, states can be annotated with atomic propositions AP by a labeling function $\ell : S \to \mathcal{P}(AP)$. Its extension to infinite words $\ell : S^\omega \to \mathcal{P}(AP)^\omega$ is a measurable function when the Borel σ-algebra of the space of words is considered in both sides, since clearly $\ell^{-1}(\{u\}\mathcal{P}(AP)^\omega) = \{w \in S^* : \ell(u) = w\}S^\omega$ is measurable for any $u \in \mathcal{P}(AP)^\omega$. Hence, if $(S^\omega, \mathcal{B}(S^\omega), \mu)$ is the probability space of executions discussed in Section 2.7.1, the set of propositional traces $(\mathcal{P}(AP)^\omega, \mathcal{B}(\mathcal{P}(AP)^\omega), \mu')$ is a probability space as well with $\mu' = \mu \circ \ell^{-1}$. For classical logics and discrete-time Markov chains, this determines the probability that a formula is satisfied. Linear-time properties φ designate subsets of executions $L(\varphi) \subseteq \mathcal{P}(AP)^\omega$, as explained in Section 2.3, so the probability of φ being satisfied is exactly $\mu'(L(\varphi))$. This is only well-defined when $L(\varphi)$ is measurable, but any ω-regular language is [HR86; Lemma 5.10] and so any LTL property holds with a well-defined probability. Branching-time formulae in CTL and CTL* can also be assigned a probability, by considering the path probabilities when evaluating each quantifier.

Model-checking techniques for probabilistic systems can be classified in two variants:

- *Probabilistic* model checking [CHV⁺18; §28] is based on the exhaustive exploration of the state space as in the quantitative case. Numerical algorithms are used to calculate the exact measure of the paths satisfying a formula or the value of other quantitative properties.

- *Statistical* model checking [AP18] is based on simulations and the Monte Carlo method. The state space is not explored exhaustively, but sampled instead according to the probability distributions on the transitions, and numerical results are estimated from these executions. For unbounded state spaces including continuous variables like times delays, this approach should be more convenient.

Among the first methods, an adaptation of the automata-theoretic approach with a deterministic Rabin automata can be used to obtain the probability of an LTL formula. Its complexity is polynomial on the size of the Markov chain and doubly exponential on the size of the formula in its most common form, although the exponential can be reduced to a single one with alternative techniques [CY95]. There are also algorithms for calculating the maximum and minimum probabilities of a formula in a Markov decision process, without actually exploring all possible ways of resolving the nondeterminism. A variation of the algorithm for LTL on DTMCs can be used to compute them, but the doubly exponential cannot be avoided.

Probabilistic temporal logics include additional operators adapted to the probabilistic setting. PCTL [HJ94] is an extension of CTL including an operator $\mathbf{P}_I$ with an interval $I \subseteq [0,1]$ in place of the universal and existential quantifiers. A formula $\mathbf{P}_I\,\varphi$ is satisfied if the probability of the paths in which the path formula φ is satisfied is in I. Syntactically, I is often written as the relation $>$, $\geq$, $<$, or $\leq$ followed by a number. An algorithm based on a bottom-up traversal of the PCTL formula can be used to check these properties. In the same way CTL* is an extension of CTL, PCTL* [BK08] is an extension of PCTL where the requirement of preceding any temporal operator by the $\mathbf{P}_I$ quantifier is dropped. Temporal operators annotated with time intervals like $\Diamond^{\leq 2}$ and $\mathbf{U}^{[4,6]}$ from real-time logics like TCTL [ACD90] are often used in probabilistic formulae. They mean that the classical property should be satisfied within the specified period, expressed in numbers of steps for discrete-time systems. They can be seen as syntactic sugar as follows

$$\varphi_1\,\mathbf{U}^{[n,m]}\,\varphi_2 \coloneqq \varphi_1\,\mathbf{U}\,\varphi_2 \;\wedge \bigvee_{n\leq k\leq m} \bigcirc^k\,\varphi_2$$

where $\bigcirc^k$ symbolizes k consecutive applications of $\bigcirc$. The quantified *until* and *always* operators are derived from it as usual.

The probabilistic model-checking techniques mentioned in this section are available in tools like PRISM [KNP11] and Storm [HJK+21]. Statistical model checking is supported for instance by the VeStA [SV13a] tools, which allow checking properties expressed in PCTL and the Quantitative Temporal Expressions (QuaTEx) logics.

Part II

Strategies and model checking

Chapter 3

The Maude strategy language

In this chapter, we describe the Maude strategy language currently available in Maude 3 and our contribution to its development. This language to control rule rewriting was designed long before this thesis by Steven Eker, Narciso Martí Oliet, José Meseguer, and Alberto Verdejo on the basis of earlier strategy languages like ELAN and Stratego (introduced in Section 2.2.3 and compared in Section 3.8) and on the previous experience with internal strategies in Maude. It was first implemented by Alberto Verdejo in Maude itself as an extension of Full Maude, which was used and tested with several examples. Shortly after, Steven Eker started its incorporation to the Maude interpreter for a better performance and integration with the rest of the Maude system. However, this implementation was not complete when we started this thesis and some interesting features were pending like strategy modules, strategy calls and recursive strategies, strategy theories and parameterization, subterm rewriting operators, and its reflection at the metalevel. We have completed this implementation, evaluated it with more examples, and developed the foundations of the language. In this sense, we provide a complete denotational semantics of the language in Section 3.4 that describes which terms result from a strategy computation, a nondeterministic small-step operational semantics in Section 3.5 to characterize the rewriting paths denoted by a strategy expression, and we discuss the expressivity of the language in Section 3.6. This chapter concludes with an overall description of the C++ implementation in Section 3.7, and a comparison with other strategy languages in Section 3.8.

Model checking of rewriting systems controlled with this language is discussed in Chapter 4. However, some of the sections of this chapter lay the groundwork for this. This strategy language is extended with probabilistic operators in Chapter 5, and examples of strategy specifications are included in Part III.

In order to illustrate the explanations of the chapter with examples, let us introduce the following Maude specification of the 15-puzzle, a classical sliding puzzle. The following functional module specifies the puzzle board, which consists of 15 sliding numbered square tiles lying in a 4×4 frame, as depicted in Figure 3.1. Solving the puzzle is obtaining the left board from a given configuration like the right one by succesively moving a tile to the blank. The puzzle is represented as a semicolon-separated list of rows, each a list of tiles, which are in turn either a natural number or a blank `b`.

```
fmod 15PUZZLE-BOARD is
  protecting NAT .

  sorts Tile Row Puzzle .
  subsorts Nat < Tile < Row < Puzzle .

  op b : -> Tile [ctor] .
  op nil : -> Row [ctor] .
  op __  : Row Row -> Row [ctor assoc id: nil prec 25] .
```

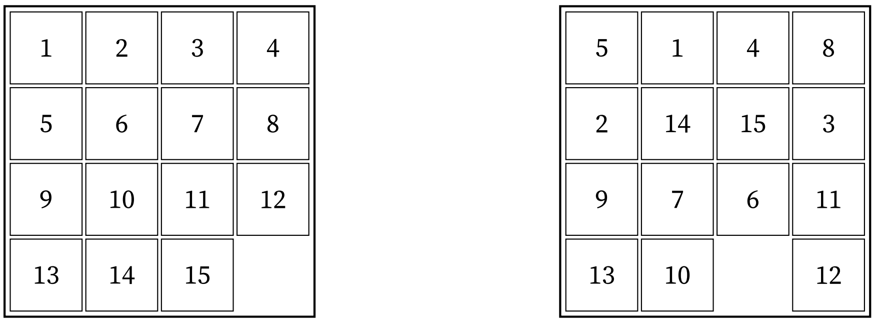


Figure 3.1: The 15-puzzle in its goal position and in a shuffled one.

```
  op _;_ : Puzzle Puzzle -> Puzzle [ctor assoc] .

  var T : Tile . var R : Row .

  op size : Row -> Nat .
  eq size(nil) = 0 .
  eq size(T R) = size(R) + 1 .
endfm
```

Moving any adjacent tile to the blank are the possible moves of the game. There is a rule for each possible direction in the following system module.

```
mod 15PUZZLE is
  protecting 15PUZZLE-BOARD .

  var T : Tile . vars LU RU LD RD : Row . var P : Puzzle .

   rl [left]  : T b => b T .
   rl [right] : b T => T b .
  crl [down]  : (LU b RU) ; (LD T RD)
             => (LU T RU) ; (LD b RD) if size(LU) = size(LD) .
  crl [up]    : (LU T RU) ; (LD b RD)
             => (LU b RU) ; (LD T RD) if size(LU) = size(LD) .
endm
```

The data representation of the square is not intended to be efficient, as shown by the conditional rules for `down` and `up`, but just to be easily readable and illustrative for the examples in this chapter. The game is discussed in more detail and strategies are proposed to solve it in Section 6.6.

3.1 Informal description

The main component of the Maude strategy language is the selective application of rules, and they can be combined with some other operators to build more complex strategic programs. In this section, the syntax and the informal semantics of these constructors are presented. Strategies are executed in the Maude interpreter using the command `srewrite` t `using` α that rewrites t according to α and shows all terms obtained at the end of this controlled but not necessarily deterministic rewriting process. Accordingly, strategies will be described by the results they produce, as transformations from a term to a set of terms. We call the term where strategies are applied the *subject term*. Strategy modules, the modules where strategies can be named and recursive strategies can be defined, are described in Section 3.2. A detailed formal semantics of what is here informally described can be found in Section 3.4.

3.1.1 Idle and fail

The simplest strategies are the constants **`idle`** and **`fail`**.

$\langle \mathit{Strat} \rangle$ ::= `idle` | `fail`

The **`fail`** strategy always *fails* for any given state, in the sense that it does not produce any result. On the contrary, the **`idle`** constant always succeeds, and returns the given state unaltered as its only result.

```
Maude> srewrite 1 b 2 using idle .

Solution 1
rewrites: 0
result Row: 1 b 2

No more solutions.
rewrites: 0

Maude> srewrite 1 b 2 using fail .

No solution.
rewrites: 0
```

In a broader sense, we say that a strategy α *fails* in a state t if it does not produce any result.

3.1.2 Rule application

Since strategies control rewriting, the basic building block of the language is the application of rules. Rules are selected by their labels, but a finer control on how they are applied is possible using an initial ground substitution, which can optionally be provided between brackets. Moreover, in order to apply a conditional rule with rewriting conditions, strategies must be provided to control them.

$\langle \mathit{Strat} \rangle$ ::= $\langle \mathit{RuleApp} \rangle$
  | `top(` $\langle \mathit{RuleApp} \rangle$ `)`

$\langle \mathit{RuleApp} \rangle$ ::= $\langle \mathit{Label} \rangle$ [`[` $\langle \mathit{Substitution} \rangle$ `]`] [`{` $\langle \mathit{StratList} \rangle$ `}`]

$\langle \mathit{Substitution} \rangle$ ::= $\langle \mathit{Variable} \rangle$ `<-` $\langle \mathit{Term} \rangle$
  | $\langle \mathit{Substitution} \rangle$ `,` $\langle \mathit{Substitution} \rangle$

$\langle \mathit{StratList} \rangle$ ::= $\langle \mathit{Strat} \rangle$
  | $\langle \mathit{Strat} \rangle$ `,` $\langle \mathit{StratList} \rangle$

The strategy expression $\mathit{label}\,[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]\{\alpha_1, \ldots, \alpha_m\}$ denotes the application anywhere within the subject term of any rule labeled *label* in the current module having exactly m rewriting condition fragments. Moreover, the variables in both rule sides and in its condition are previously instantiated by the substitution that maps x_i to t_i for all $1 \leq i \leq n$.

```
Maude> srewrite 1 b 2 ; 3 b 4 using right .

Solution 1
rewrites: 1
```

```
result Puzzle: 1 2 b ; 3 b 4

Solution 2
rewrites: 2
result Puzzle: 1 b 2 ; 3 4 b

No more solutions
rewrites: 2

Maude> srewrite 1 b 2 ; 3 b 4 using left[T <- 1] .

Solution 1
rewrites: 1
result Puzzle: b 1 2 ; 3 b 4

No more solutions
rewrites: 1
```

When strategies are specified within curly brackets, each α_i controls the rewriting of the i-th rewriting condition fragment $l_i \Rightarrow r_i$ of the selected rule. As usual when evaluating rule conditions, both sides l_i and r_i are instantiated with the substitution determined by the matching of the lefthand side and the previous condition fragments. However, l_i is rewritten according to the strategy α_i, and only its solutions are matched against r_i. As usual again, the evaluation of the next rewriting fragments continues with every result of the match, yielding potentially different one-step rewrites. We introduce here a simple module to exemplify the application of rules with rewriting fragments. Even though the `move` rule is marked `nonexec`, because of the free variable `M`, it can be executed by a rule application that instantiates such a variable.

```
mod 15PUZZLE-LOG is
  protecting 15PUZZLE .
  protecting LIST{Qid} .

  sort PuzzleLog .
  op <_|_> : List{Qid} Puzzle -> PuzzleLog [ctor] .

  var M : Qid . var L : List{Qid} . vars P P' : Puzzle .

  crl [move] : < L | P > => < L M | P' > if P => P' [nonexec] .
endm

Maude> srewrite < nil | 1 b 2 > using move[M <- 'left]{left} .

Solution 1
rewrites: 3
result PuzzleLog: < 'left | b 1 2 >

No more solutions.
rewrites: 3
```

A special strategy for rule application is the constant **`all`** that executes any available rule in the module, labeled or not. Rewriting conditions are evaluated without restrictions, as in the usual `rewrite` command. However, rules marked with `nonexec` and the implicit rules that handle external objects (see Section 2.5.3) are excluded.

```
Maude> srewrite < nil | 1 b 2 > using all .
```

```
Solution 1
rewrites: 3
result PuzzleLog: < nil | b 1 2 >

Solution 2
rewrites: 4
result PuzzleLog: < nil | 1 2 b >

No more solutions.
rewrites: 4
```

Rules are usually applied anywhere in the subject term; but rule application expressions can be prefixed by the **top** modifier to restrict matching to the top of the subject term. In combination with the subterm rewriting operator, to be explained in Section 3.1.6, this allows restricting rewriting to specific positions within the term structure. Nevertheless, **top**(α) is not necessarily deterministic, because the rule's lefthand side matching takes place modulo structural axioms like associativity and commutativity. Additionally, rules can contain matching and rewriting condition fragments that may produce multiple substitutions. For example, if we had a rule `multimv` as `LU b RU` ⇒ `LU RU b` that jumps multiple positions at once, we would obtain:

```
Maude> srewrite 1 b 2 b 3 using top(multimv) .

Solution 1
rewrites: 1
result Row: 1 2 b 3 b

Solution 2
rewrites: 2
result Row: 1 b 2 3 b

No more solutions.
rewrites: 2
```

3.1.3 Tests

Tests can be used to check properties of the subject term and then abandon those rewriting paths in which they are not satisfied. Since matching is one of the most basic features of rewriting, tests are based on matching and on evaluation of equational conditions.

⟨*EqCondition*⟩ ::= ⟨*BoolTerm*⟩
  | ⟨*Term*⟩ = ⟨*Term*⟩
  | ⟨*Term*⟩ := ⟨*Term*⟩
  | ⟨*EqCondition*⟩ /\ ⟨*EqCondition*⟩

⟨*Test*⟩ ::= `amatch` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩]
  | `match` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩]
  | `xmatch` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩]

Three variants of tests are available regarding where matching takes place: **match** tries to match at the top of the subject term, **amatch** matches anywhere, and **xmatch** matches only at the top, but *with extension* modulo the structural axioms of the top symbol. Their behavior is equivalent to an **idle** when the test passes, and to a **fail** when it does not. In other words, the test strategy applied to a term t will evaluate to $\{t\}$ if the matching and condition evaluation succeeds, and to $\varnothing$ if they fail.

```
Maude> srewrite 1 b 2 using xmatch b N s.t. N =/= 1 .

Solution 1
rewrites: 1
result Row: 1 b 2

No more solutions.
rewrites: 1
```

If the pattern and the condition contain variables that are bound in the current scope, they will be instantiated before matching. The condition is evaluated like an equational condition (see Section 2.5).

3.1.4 Regular expressions

The first strategy combinators to describe execution paths are the typical regular expression constructors.

$$
\begin{aligned}
\langle \mathit{Strat} \rangle ::= {} & \langle \mathit{Strat} \rangle \texttt{ ; } \langle \mathit{Strat} \rangle \\
\mid \quad & \langle \mathit{Strat} \rangle \texttt{ | } \langle \mathit{Strat} \rangle \\
\mid \quad & \langle \mathit{Strat} \rangle \texttt{ * } \\
\mid \quad & \langle \mathit{Strat} \rangle \texttt{ + }
\end{aligned}
$$

The *concatenation* $\alpha \,;\, \beta$ evaluates α and then applies β to every result yielded by α; it then returns the collection of all results from β. The *alternation* combinator $\alpha \,|\, \beta$ evaluates α or β nondeterministically, so it returns the results of both strategies.

```
Maude> srewrite 1 2 ; 3 b using left ; up | up ; left .

Solution 1
rewrites: 24
result Puzzle: b 2 ; 1 3

Solution 2
rewrites: 32
result Puzzle: b 1 ; 3 2

No more solutions.
rewrites: 32
```

The *iteration* $\alpha\,*$ executes α zero or more times consecutively. It can be described recursively as $\alpha\,* \equiv$ `idle` | α ; $\alpha\,*$.

```
Maude> srewrite 1 b 2 3 using right * .

Solution 1
rewrites: 0
result Row: 1 b 2 3

Solution 2
rewrites: 1
result Row: 1 2 b 3

Solution 3
rewrites: 2
result Row: 1 2 3 b
```

```
No more solutions.
rewrites: 2
```

The already seen **idle** and **fail** constants complete the regular expressions sublanguage. The usual notation for the non-empty iteration α + is also available in the language as a derived combinator α + ≡ α ; α *. Following the usual convention, the iteration operators have greater precedence than the concatenation for parsing, which in turn has more precedence than the alternation.

3.1.5 Conditional and derived operators

The next strategy combinator is the typical conditional or *if-then-else* construct. However, its condition is a strategy too, which is considered satisfied if it provides at least a result. This idea has been borrowed from Stratego and ELAN (see Section 3.8).

$\langle Strat \rangle$ ::= $\langle Strat \rangle$? $\langle Strat \rangle$: $\langle Strat \rangle$

More precisely, the evaluation of the expression α ? β : γ starts by the condition α. Any of its results is continued with the positive branch β as if $\alpha;\beta$ were run. But if α fails, the negative branch γ is evaluated instead on the initial term. In case α is a test, the conditional behaves like a typical Boolean conditional.

```
Maude> srewrite 1 b 2 3 4 using right * ; (right ? fail : idle) .

Solution 1
rewrites: 6
result Row: 1 2 3 4 b

No more solutions.
rewrites: 6
```

Using the conditional strategy, multiple useful derived combinators can be defined and are available in the language:

$\langle Strat \rangle$ ::= $\langle Strat \rangle$ or-else $\langle Strat \rangle$
  | not($\langle Strat \rangle$)
  | $\langle Strat \rangle$!
  | try($\langle Strat \rangle$)
  | test($\langle Strat \rangle$)

The combinator α **or-else** β evaluates to the result of α unless α fails; in that case, it evaluates to the result of β. Consequently, it is equivalent to

$$\alpha \ \texttt{or-else}\ \beta \ \equiv\ \alpha \ \texttt{?}\ \texttt{idle} \ \texttt{:}\ \beta$$

The negation combinator **not**(α) is defined as **not**(α) ≡ α ? **fail** : **idle**, thus reversing the binary result of evaluating α. The *normalization* operator α ! evaluates a strategy repeatedly until just before it fails.

$$\alpha \ ! \ \equiv\ \alpha \ * \ ; \ \texttt{not}(\alpha)$$

The strategy **try**(α) works as if α were evaluated, but when α fails it results in $\{t\}$ instead, where t was the initial subject term.

$$\texttt{try}(\alpha) \ \equiv\ \alpha \ \texttt{?}\ \texttt{idle} \ \texttt{:}\ \texttt{idle}$$

Finally, **test**(α) checks the binary outcome of α, by succeeding without changing the term if α succeeds and failing otherwise.

$$\texttt{test}(\alpha) \ \equiv\ \texttt{not}(\alpha) \ \texttt{?}\ \texttt{fail} \ \texttt{:}\ \texttt{idle}$$

Notice that this is the same as **not**(**not**(α)).

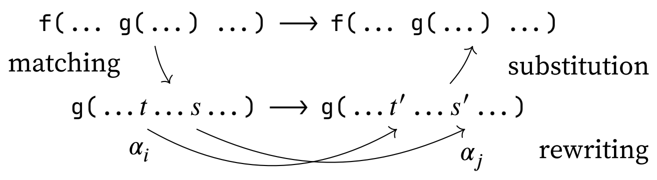

f(... g(...) ...) ⟶ f(... g(...) ...)
matching
substitution
g(...t...s...) ⟶ g(...t′...s′...)
αi
αj
rewriting

Figure 3.2: Behavior of the `amatchrew` combinator.

3.1.6 Rewriting of subterms

The following family of operators allows rewriting selected subterms using some given strategies. Additionally, it can be used to extract information from the subject term and use it to condition the strategy execution.

⟨*Strat*⟩ ::= `amatchrew` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩] `by` ⟨*VarStratList*⟩
  | `matchrew` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩] `by` ⟨*VarStratList*⟩
  | `xmatchrew` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩] `by` ⟨*VarStratList*⟩

⟨*VarStratList*⟩ ::= ⟨*VarId*⟩ `using` ⟨*Strat*⟩
  | ⟨*VarStratList*⟩ `,` ⟨*VarStratList*⟩

The subterms to be rewritten are selected by matching against a pattern. A subset of its variables hold the positions of these subterms and they are bound to a strategy each.

$$\texttt{matchrew}\ P[x_1, \ldots, x_n, x_{n+1}, \ldots, x_m]\ \texttt{s.t.}\ C\ \texttt{by}\ x_1\ \texttt{using}\ \alpha_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{using}\ \alpha_n$$

These variables must be distinct, but they may appear more than once in the pattern. The strategy explores all possible matches of the pattern P for the subject term that satisfy the condition C. For each, it extracts the subterms that match the variables x_1 to x_n, and rewrites them separately in parallel using their corresponding strategies α_1 to α_n. Every combination of their results is reassembled in the original term in place of the original subterms. Like for the tests, three variants can be selected by changing the initial keyword: `matchrew` for matching on top, `xmatchrew` for doing so with extension, and `amatchrew` for matching everywhere. The behavior of the `amatchrew` operator is illustrated in Figure 3.2.

```
Maude> srewrite 1 b 2 ; 3 b 4 using matchrew RU ; RD
                                    by RU using left, RD using right .

Solution 1
rewrites: 2
result Puzzle: b 1 2 ; 3 4 b

No more solutions.
rewrites: 2
```

Variables bound in the outer scope are instantiated in the pattern P and condition C, like in tests. Moreover, the variable scope of the substrategies $\alpha_1, \ldots, \alpha_n$ is extended with the variables in P and C. Notice that `matchrew` and its variants are the only strategies that define static variable scopes, and the only that bind variables along with strategy calls.

3.1.7 Pruning of alternative solutions

The strategy language combinators are nondeterministic in different ways: rule applications may produce multiple rewrites, the alternation operator $\alpha \,|\, \beta$ could choose α or β, the iteration $\alpha\,*$ may execute α any number of times, and `matchrew` could start with different matches. The default behavior of the strategy rewriting engine is to do an exhaustive search on all the allowed rewriting paths, and show all reachable results. However, the user may be interested in any solution without distinction, or may know that a single solution exists. In these cases and for efficiency, the `one` combinator is available.

⟨*Strat*⟩ ::= `one(` ⟨*Strat*⟩ `)`

`one`(α) evaluates α in the given term. If α fails, so does `one`(α). Otherwise, a solution for α is chosen nondeterministically as its single result. However, since the purpose of introducing this strategy is efficiency, the chosen solution will be the first that is found.

```
Maude> srewrite 1 b 2 3 using one(right +) .

Solution 1
rewrites: 1
result Row: 1 2 b 3

No more solutions.
rewrites: 1
```

More meaningful usage examples of `one` can be found in Section 6.4. ELAN [BKK⁺01] counts with similar constructs `dc one` and `first one`, whose behaviors coincide with our `one` when a single strategy argument is provided (see Section 3.8).

3.1.8 Strategy calls

Instead of writing huge monolithic strategy expressions and copying them verbatim to be executed, complex strategies are better split in several expressions referred to by meaningful names. These strategies are defined in strategy modules, which will be explained in Section 3.2, but also extend the language of expressions with a *strategy call* constructor.

⟨*Strat*⟩ ::= ⟨*StratCall*⟩

⟨*StratCall*⟩ ::= ⟨*StratId*⟩ `(` ⟨*TermList*⟩ `)`
  | ⟨*StratId*⟩

Strategies are invoked by a strategy label, and may receive input arguments of any sort in the module, according to their declarations. These parameters are written in a comma-separated list between parentheses. For strategies without parameters, the parentheses can be omitted, except when there is a rule with the same label in the module.

3.1.9 Strategy commands

Strategies are evaluated in the interpreter using the `srewrite` and `dsrewrite` commands, which look for solutions of the strategy and show all they find. Moreover, the standard `search` command is extended to control the search space by a strategy expression.

⟨*Commands*⟩ ::= `srewrite` [`[` ⟨*Nat*⟩ `]`] [`in` ⟨*ModId*⟩] ⟨*Term*⟩ `using` ⟨*Strat*⟩ `.`
  | `dsrewrite` [`[` ⟨*Nat*⟩ `]`] [`in` ⟨*ModId*⟩] ⟨*Term*⟩ `using` ⟨*Strat*⟩ `.`
  | `search` [⟨*DoubleBound*⟩] [`in` ⟨*ModId*⟩] ⟨*Term*⟩ ⟨*Arrow*⟩ ⟨*Term*⟩
  [`s.t.` ⟨*EqCondition*⟩] `using` ⟨*Strat*⟩ `.`

Following the usual convention of Maude commands, an optional bound on the number of solutions to be calculated can be specified between brackets after the command keyword. Nevertheless, more solutions can be requested afterwards using the `continue` command. The strategy will be evaluated in the current Maude module, but a different module could be specified preceded by `in`. The `srewrite` keyword can be abbreviated to `srew`, and `dsrewrite` to `dsrew`.

The `srewrite` command explores the rewriting tree using a *fair* policy that ensures that all solutions are eventually found if there is enough memory. Not being completely a breadth-first search, it may explore multiple execution paths in parallel. More details can be found in Section 3.7. On the contrary, the `dsrewrite` command performs a depth-first exploration of the tree. It is usually faster and uses less memory, but some solutions may not be reached because of nonterminating execution branches.

```
Maude> srew [2] b 1 2 ; b 3 using right + .

Solution 1
rewrites: 2
result Puzzle: 1 b 2 ; b 3

Solution 2
rewrites: 3
result Puzzle: b 1 2 ; 3 b

Maude> dsrew [2] b 1 2 ; b 3 using right + .

Solution 1
rewrites: 1
result Puzzle: 1 2 b ; 3 b

Solution 2
rewrites: 2
result Puzzle: 1 b 2 ; b 3
```

Notice that the order in which solutions are obtained may differ. The displayed rewrite count reflects all the equational and rule rewrites that have been applied until the solution is found, but its origin could be in other execution branches not yet completed or abandoned because they do not lead to a solution.

The search conducted by the `srewrite` and `dsrewrite` commands theoretically explores the subtree of the rewriting tree pruned by the restrictions of the strategy, but their search space is actually a graph. The execution engine is able to detect already visited execution states, thus preventing the redundant evaluation of the same strategy on the same term. Consequently, the strategy evaluation may finish in situations where operationally nonterminating executions are involved.

```
Maude> srew 1 b using (left | right) * .

Solution 1
result Row: 1 b

Solution 2
result Row: b 1

No more solutions.
```

However, strategy evaluation is not always terminating, since the underlying rewriting system may have infinitely many states. In addition, the cycle detector does not operate for strategy calls unless they are tail recursive and do not have parameters.

The previous commands only show the solutions of the strategy evaluation, but no other states reachable by rewriting using it. These can be found using the standard `search` command with an additional `using` suffix followed by a strategy expression. Like in the standard command, the user should provide the initial term of the search, the pattern and condition that select the solution to be shown, and the type of search using the arrows $\Rightarrow 1$ for solutions reachable after a single rewrite, `=>*` for solutions after zero or more rewrites, and `=>+` for terms found after at least one rewrite. Moreover, bounds for the number and maximum depth of the solutions can be specified with two numbers between brackets at the beginning of the command.

```
Maude> search 1 2 b =>+ P:Puzzle using left ; left .

Solution 1 (state 1, next strategy: left)
states: 3  rewrites: 2
P:Puzzle --> 1 b 2

Solution 2 (state 2)
states: 3  rewrites: 2
P:Puzzle --> b 1 2

No more solutions.
states: 3  rewrites: 2
```

Like in the uncontrolled search, a path to a given solution can be recovered with the `show path` and `show path labels` commands followed by the state number that appears in the `search` output.

```
Maude> show path 2 .
state 0, NeRow: 1 2 b
===[ rl T b => b T [label left] . ]===>
state 1, NeRow: 1 b 2
===[ rl T b => b T [label left] . ]===>
state 2, NeRow: b 1 2
```

The portion of the strategy-controlled rewrite graph explored by the command can be shown with the `show search graph` command.

```
Maude> show search graph .
state 0, NeRow: 1 2 b
arc 0 ===> state 1 (rl T b => b T [label left] .)

state 1, NeRow: 1 b 2
arc 0 ===> state 2 (rl T b => b T [label left] .)

state 2, NeRow: b 1 2
```

The benefits of these features are not observed in this simple example where the intermediate states can be easily inferred from the strategy expression, but they can clarify the execution of more complex strategies and solve restricted reachability problems.

Another arrow type `=>!` is admitted in the strategy-controlled search to limit the solutions of this command to solutions of the strategy. In fact, the command `search` t `=>!` P `s.t.` C `using` α is equivalent to `srewrite` t `using` α ; **`match`** P **`s.t.`** C, except that the first one also prints the matching substitution and allows obtaining the additional information we have seen.

```
Maude> search 1 2 b =>! P:Puzzle using left ; left .

Solution 1 (state 2)
states: 3  rewrites: 2
P:Puzzle --> b 1 2

No more solutions.
states: 3  rewrites: 2
```

In the standard search, this arrow has a different meaning and restricts solutions to irreducible terms. Terms can also be said to be irreducible with respect to a strategy, but this concept is less clear with memoryful strategies for which a term may be irreducible in an execution path and reducible in another. For instance, this is the case of `1 b 2` from `1 2 b` using the `left | left ; left` strategy.

Searching under the control of a strategy is more subtle than searching in the unrestricted rewrite graph. The concept of reachable state by strategy-controlled rewriting is aware of the possibility that strategies may fail at some point in the execution, discarding the previous steps. In this sense, strategy expressions can sometimes be misleading.

```
Maude> search 1 b =>* P:Puzzle using left ; fail .
search in 15PUZZLE-MAIN : 1 b =>* P:Puzzle using left ; fail .

No solution.
states: 2  rewrites: 1
```

This search does not provide `1 b` or `b 1` as a result, although they must be visited by the execution engine before discarding the complete rewriting path with the **`fail`** strategy. Consequently, the `search` implementation cannot know whether a visited term is actually valid until it has found a solution or infinite execution through that term. This means that validating a single rewrite might imply an unbounded and potentially nonterminating exploration of the search space. An explicit definition of which are the intermediate states of a strategy execution is discussed in Section 3.5, and the details about the implementation of the command can be found in Section 4.7. This search is carried out in the same graph used by the strategy-controlled model checker.

3.2 Strategy modules and recursion

Callable strategies can be declared and defined in strategy modules, as anticipated in Section 3.1.8. Apart from the benefits already described, this increases the expressiveness of the strategy language by means of recursive and mutually recursive strategies, which can also keep a control state in their parameters.

Strategy modules are a third level of Maude modules, devoted to representing the control of rewriting systems by means of the strategy language, as the classical functional and system modules were dedicated to represent equational and rewrite theories, respectively. They are introduced by the **`smod`** keyword and closed by **`endsm`**.

⟨*Module*⟩ ::= `smod` ⟨*ModId*⟩ [⟨*ParameterList*⟩] `is` ⟨*SmodElt*⟩* `endsm`

⟨*SmodElt*⟩ ::= ⟨*ModElt*⟩
  | ⟨*StratDecl*⟩
  | ⟨*StratDef*⟩

Strategy modules are extensions of system modules in the same way system modules are extensions of functional modules. Therefore, they may include any declaration or statement that is allowed in these lower-level modules. However, to promote a clean separation between the rewrite theory specification and its control, including only strategy-related statements in strategy modules is encouraged, apart from importations and variable declarations. Only strategy modules are able to import other strategy modules, but they can import modules of any kind using the usual statements: `including`, `extending`, and `protecting`. The semantic difference between these importation modes is described in [CDE+20; §10.2.1].

The strategy-related statements are strategy declarations and strategy definitions.

⟨*StratDecl*⟩ ::= `strat` ⟨*SLabel*⟩+ [`:` ⟨*Type*⟩*] `@` ⟨*Type*⟩ `.`

$\langle StratDef \rangle$ ::= `sd` $\langle StratCall \rangle$ `:=` $\langle Strat \rangle$ `.`
  | `csd` $\langle StratCall \rangle$ `:=` $\langle Strat \rangle$ `if` $\langle Condition \rangle$ `.`

The following line declares a strategy *slabel* that receives n parameters of sorts $s_1, \ldots, s_n$, and that is intended to control rewriting of terms of sort s.

strat *slabel* : s_1 ... s_n @ s .

Many strategies with a common signature can be defined in the same declaration, by writing multiple identifiers, in which case the plural keyword **strats** is preferred. The input parameter sorts and the colon can be omitted if the strategy has no parameters.

Strategies are defined by means of conditional or unconditional strategy definitions.

sd *slabel*(p_1, ..., p_n) := α .
csd *slabel*(p_1, ..., p_n) := α **if** C .

These definitions associate the strategy name *slabel* to an expression α whenever the input parameters match the patterns $p_1, \ldots, p_n$. The syntax of conditions is the same as that of equations and tests, explained in Section 3.1.3. The lefthand sides of the definitions are strategy calls as described in Section 3.1.8. Variables in the pattern and the condition may appear in the strategy expression α. When a strategy is called, all strategy definitions that match the input arguments will be executed and the union of all their results is the result of the call. Hence, strategies without any definition at all behave like a **fail**.

The following is an example of strategy module that imports the system module `15PUZZLE`, declares two strategies, `loop` and `move`, and gives definitions for them.

```
smod 15PUZZLE-STRATS is
  protecting 15PUZZLE .
  protecting INT .

  strat loop           @ Puzzle .
  strat move : Int Int @ Puzzle .

  var N : Nat . var M : Int .

  sd loop := left ; up ; right ; down .

  sd move(0, 0)      := idle .
  sd move(s(N), M)   := right ; move(N, M) .
  sd move(- s(N), M) := left  ; move(- N, M) .
  sd move(0, s(N))   := down  ; move(0, N) .
  sd move(0, - s(N)) := up    ; move(0, - N) .
endsm
```

While `loop` simply displaces the blank in a fixed loop, `move` moves it some offset in the board. Its five definitions have disjoint patterns, so that only a single definition will be executable for any given input, so the strategy is deterministic. If `M` were written instead of `0` in the last two definitions, multiple definitions could be activated for the same call, and both vertical and horizontal displacements would be mixed nondeterministically to bridge the distance.

```
Maude> srewrite 1 2 3 ; 4 5 6 ; 7 b 8 using move(1, -2) .

Solution 1
rewrites: 70
result Puzzle: 1 2 b ;
               4 5 3 ;
               7 8 6
```

```
No more solutions.
rewrites: 70
```

3.2.1 Parameterization

Maude's support for parameterized programming in its functional and system modules (see Section 2.5.1) has been extended to strategy modules likewise. Strategy modules can be parameterized by functional and system theories too, but also by formal strategies expressed as *strategy theories*.

⟨*Theory*⟩ ::= `sth` ⟨*ModId*⟩ `is` ⟨*SmodElt*⟩* `endsth`

Strategy theories are syntactically identical to strategy modules. They are allowed to contain any statement, although strategy declarations and definitions are mainly expected, and to include functional or system theories using the **`including`** mode (or modules using any importation mode). The formal strategies declared in the theory should be realized by the actual target modules. Modules other than strategy modules cannot be parameterized by strategy theories. A simple example of strategy theory is the following `STRIV` that declares a single strategy without arguments operating on the formal sort of the `TRIV` theory.

```
sth STRIV is
  including TRIV .
  strat st @ Elt .
endsth
```

Parameterized by this theory, we can specify the following module `REPEAT`, which defines a strategy `repeat(`*n*`)` that applies *n* times the parameter strategy `st`.

```
smod REPEAT{X :: STRIV} is
  protecting NAT .

  strat repeat : Nat @ X$Elt .

  var N : Nat .

  sd repeat(0)    := idle .
  sd repeat(s(N)) := st ; repeat(N) .
endsm
```

Again, views are required to interpret strategy theories and instantiate strategy modules. Thus, the syntax of views is extended to support strategy mappings:

⟨*ViewElt*⟩ ::= `strat` ⟨*StratId*⟩ `to` ⟨*StratId*⟩ `.`
  | `strat` ⟨*StratId*⟩ [`:` ⟨*Type*⟩ *] `@` ⟨*Type*⟩ `to` ⟨*StratId*⟩ `.`
  | `strat` ⟨*StratCall*⟩ `to expr` ⟨*Strat*⟩ `.`

Three different forms of strategy mapping are offered with the same structure as operator mappings [CDE+20; §6.3.2]: using **`strat`** *formalName* `:` s_1 `...` s_n `@` s **`to`** *actualName*, the formal strategy with the given name *formalName* and signature is mapped to an actual strategy in the target module whose name is *actualName* and whose input arguments' sorts are the translation of the formal signature as follows from the sort mappings of the view. To map all overloaded strategies with a given name at once, another mapping **`strat`** *formalName* **`to`** *actualName* is available. Finally, the inline mapping **`strat`** *slabel*$(x_1, \dots, x_n)$ **`to expr`** α maps the strategy *slabel* whose input types are those of $x_1, \dots, x_n$ to the strategy expression α that may depend on these variables. Only variables are allowed as arguments in the lefthand side.

For instance, we can repeatedly apply the `loop` strategy defined in the strategy module `15PUZZLE-STRATS` at the beginning of this section, by instantiating the `REPEAT` module with the following view from `STRIV` to `15PUZZLE-STRATS`.

```
view Loop from STRIV to 15PUZZLE-STRATS is
  sort Elt to Puzzle .
  strat st to loop .
endv
```

`Loop` maps the `Elt` sort to the `Puzzle` sort, and the formal strategy `st` to the `loop` strategy in `15PUZZLE-STRAT`. Instantiating `REPEAT{X :: STRIV}` with the view `Loop` by writing `REPEAT{Loop}`, we can now apply `repeat` for `loop`.

```
Maude> srew 1 2 ; 3 b using repeat(2) .

Solution 1
rewrites: 64 in 0ms cpu (0ms real) (~ rewrites/second)
result Puzzle: 3  1 ;
               2  b

No more solutions.
rewrites: 64 in 0ms cpu (0ms real) (~ rewrites/second)
```

Typically, the target of a view from a strategy theory is a strategy module but, using the `to expr` mapping, views can be directly specified for system modules. For example, the following view `Right` to the system module `15PUZZLE` maps `st` to the rule application expression `right`.

```
view Right from STRIV to 15PUZZLE is
  sort Elt to Puzzle .
  strat st to expr right .
endv
```

Then, in a module including `REPEAT{Right}`, we can execute:

```
Maude> srew b 1 2 3 4 using repeat(3) .

Solution 1
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result NeRow: 1 2 3 b 4

No more solutions.
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
```

Other combinations of initial theories and target modules or theories are possible, including views from functional or system theories to strategy modules or theories, as well as parameterized views with parameters bound to strategy theories. These possibilities and their implications are discussed in [CDE+20; §6.3.2]. A more realistic example of parameterized strategy module can be seen in Section 6.1, and several others are gathered in [RMP+19c].

3.3 Reflecting strategies at the metalevel

In Section 2.5.2, we have seen that Maude is a reflective language, where functional and system modules can be treated as data, and so be transformed and manipulated within Maude itself. Many language extensions and verification tools for Maude specifications are built in this way.

The strategy language, strategy modules, and their related operations are also supported at the metalevel. In fact, all new features added to Core Maude are routinely reflected at the

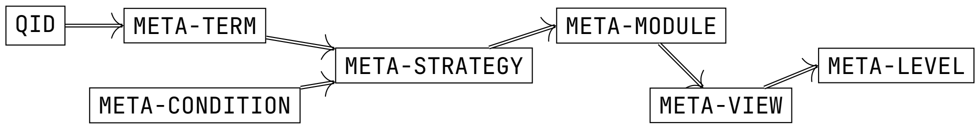


Figure 3.3: Structure of the metalevel after the inclusion of the strategy language.

metalevel to make Maude metaprogramming more powerful. Thanks to this reflective representation, new features are also translated to the extended interpreter of Full Maude, where Francisco Durán has recently incorporated the strategy language. Similarly, other interactive applications and specific frameworks built on top of Maude can benefit from having strategies represented at the metalevel, and potential formal tools can be written to reason about strategies. Another relevant consequence is that the strategy language is user-extensible at the metalevel. Hence, design decisions about which features are directly supported by the language are less dramatic than in a non-reflective setting, since the language can be extended at the user choice. Chapter 7 is devoted to procedures and examples to do so.

The Maude metalevel is a hierarchy of modules specifying the different Maude objects and operations, as illustrated in Figure 3.3. Terms are metarepresented in the `META-TERM` module as terms of sort `Term`, modules are defined in `META-MODULE` as terms of sort `Module` along with its statements, views are represented in `META-VIEW` as terms of sort `View`, and `META-LEVEL` represents operations like reduction, rule application, rewriting... as *descent functions*. Thus, to reflect the strategy language, we have specified it in a new module `META-STRATEGY`, extended `META-MODULE` and `META-VIEW` with strategy modules and views, and incorporated the strategy-rewriting operations to `META-LEVEL`. First, the strategy language constructs have been defined as follows:

```
sorts RuleApplication CallStrategy Strategy StrategyList .
subsorts RuleApplication CallStrategy < Strategy < StrategyList .

ops fail idle : → Strategy [ctor] .
op all : → RuleApplication [ctor] .
op _[_]{_} : Qid Substitution StrategyList → RuleApplication [ctor ...] .
op top : RuleApplication → Strategy [ctor] .
op match_s.t._ : Term Condition → Strategy [ctor ...] .
op _|_ : Strategy Strategy → Strategy [ctor assoc comm id: fail ...] .
op _;_ : Strategy Strategy → Strategy [ctor assoc id: idle ...] .
op _or-else_ : Strategy Strategy → Strategy [ctor assoc ...] .
op _+ : Strategy → Strategy [ctor] .
op _?_:_ : Strategy Strategy Strategy → Strategy [ctor ...] .
op matchrew_s.t._by_ : Term Condition UsingPairSet → Strategy [ctor] .
op _[[_]] : Qid TermList → CallStrategy [ctor prec 21] .
op one : Strategy → Strategy [ctor] .
*** and others (see [CDE+20; §16.3] or the Maude distribution)
```

The syntax of the representations coincides with that of the original operators except for strategy calls, which are written `_[[_]]` to avoid ambiguities. Using this metarepresentation of strategy expressions, strategy declarations and definitions are similarly represented in `META-MODULE`.

```
sorts StratDecl StratDeclSet .
subsort StratDecl < StratDeclSet .
op strat_:_@_[_]. : Qid TypeList Type AttrSet → StratDecl [ctor ...] .
op none : → StratDeclSet [ctor] .
op __ : StratDeclSet StratDeclSet
        → StratDeclSet [ctor assoc comm id: none ...] .
eq O:StratDecl O:StratDecl = O:StratDecl .
```

```
sorts StratDefinition StratDefSet .
subsort StratDefinition < StratDefSet .
op sd_:=_[_]. : CallStrategy Strategy AttrSet → StratDefinition [ctor ...] .
op csd_:=_if_[_]. : CallStrategy Strategy EqCondition AttrSet
                    → StratDefinition [ctor ...] .
*** StratDefSet defined like StratDeclSet
```

And the Module sort has also been extended with new symbols for strategy modules and theories.

```
op smod_is_sorts_._______endsm : Header ImportList SortSet SubsortDeclSet
  OpDeclSet MembAxSet EquationSet RuleSet StratDeclSet StratDefSet
  → StratModule [ctor ...] .
op sth_is_sorts_._______endsth : Header ImportList SortSet SubsortDeclSet
  OpDeclSet MembAxSet EquationSet RuleSet StratDeclSet StratDefSet
  → StratTheory [ctor ...] .
```

Similarly, the view symbol of META-VIEW has been extended with a new entry for strategy bindings of sort StratMapping:

```
op view_from_to_is___endv : Header ModuleExpression ModuleExpression
  SortMappingSet OpMappingSet StratMappingSet → View [ctor ...] .

sorts StratMapping StratMappingSet .
subsort StratMapping < StratMappingSet .

op strat_to_. : Qid Qid → StratMapping [ctor] .
op strat_:_@_to_. : Qid TypeList Type Qid → StratMapping [ctor] .
op strat_to expr_. : CallStrategy Strategy → StratMapping [ctor] .
```

Finally, the functionality of the commands srewrite and dsrewrite is metarepresented in a descent function defined in the META-LEVEL module.

```
op metaSrewrite : Module Term Strategy SrewriteOption Nat
                  ⇝ ResultPair? [special (..)] .

sort SrewriteOption .
ops breadthFirst depthFirst : → SrewriteOption [ctor] .
```

It receives the metarepresentations of a module, a term, a strategy, and the search type for the results of rewriting the term according to the strategy in that module. Since this may lead to multiple solutions, the last parameter is used to enumerate them in increasing order until the failure constant is returned. As expected from Section 3.1.9, breadthFirst corresponds to srewrite and depthFirst to dsrewrite. For example, srewrite 0 b 0 using right * at the metalevel is:

```
Maude> red in META-LEVEL : metaSrewrite(upModule('15PUZZLE, false),
  '__['0.Zero, 'b.Tile, '0.Zero], ('right[none]{empty}) *, breadthFirst, 0) .
rewrites: 2
result ResultPair: {'__['0.Zero,'b.Tile,'0.Zero],'Row}

Maude> red metaSrewrite(upModule('15PUZZLE, false),
  '__['0.Zero, 'b.Tile, '0.Zero], ('right[none]{empty}) *, breadthFirst, 1) .
rewrites: 2
result ResultPair: {'__['0.Zero,'0.Zero,'b.Tile],'Row}
```

```
Maude> red metaSrewrite(upModule('15PUZZLE, false),
  '__['0.Zero, 'b.Tile, '0.Zero], ('right[none]{empty}) *, breadthFirst, 2) .
rewrites: 2
result ResultPair: (failure).ResultPair?
```

The strategy-controlled extension of the `search` command is also represented at the metalevel in the descent functions `metaStrategySearch` and `metaStrategySearchPath`.

```
op metaStrategySearch     : Module Term Term Condition Qid Strategy Bound Nat
                            ⇝ ResultTriple? [special (...)] .
op metaStrategySearchPath : Module Term Term Condition Qid Strategy Bound Nat
                            ⇝ Trace? [special (...))] .
```

These operators are the counterparts of the classical `metaSearch` and `metaSearchPath` functions of the uncontrolled search, with identical signatures except for the additional `Strategy` argument. The first function is used to enumerate all solutions of the search using the index in the last argument, which are described by the matched term, its sort, and the matching substitution. The second function enumerates the path leading to the corresponding solutions instead of the solutions themselves, as the `show path` command does after a search. When there are no more solutions, the commands are reduced to `failure` constants. Their other arguments are the metarepresentation of the module, the initial term, the matching pattern, the condition, the arrow type among `'*`, `'+` and `'!`, and the depth bound for solutions, which can be `unbounded`. For example, the metarepresentation of `search 0 b 0 =>+ R using left *` is:

```
Maude> red metaStrategySearch(upModule('15PUZZLE, false),
           '__['0.Zero, 'b.Tile, '0.Zero], 'R:Row, nil, '+,
           ('left[none]{empty}) *, unbounded, 0) .
rewrites: 4
result ResultTriple: {'__['b.Tile,'0.Zero,'0.Zero],'NeRow,
                      'R:Row <- '__['b.Tile,'0.Zero,'0.Zero]}

Maude> red metaStrategySearch(upModule('15PUZZLE, false),
           '__['0.Zero, 'b.Tile, '0.Zero], 'R:Row, nil, '+,
           ('left[none]{empty}) *, unbounded, 1) .
rewrites: 3
result ResultTriple?: (failure).ResultTriple?
```

Parsing and pretty printing strategy expressions are also available at the metalevel in the functions `metaParseStrategy` and `metaPrettyPrintStrategy`, whose counterparts for terms are `metaParse` and `metaPrettyPrint`.

```
op metaParseStrategy : Module VariableSet QidList ⇝ Strategy? [special (...)] .
op metaPrettyPrintStrategy : Module VariableSet Strategy PrintOptionSet
                             ⇝ QidList [special (...)] .
```

The `metaParseStategy` function receives a list of tokens in the form of quoted identifiers and produces the metarepresentation of the strategy expression that can be parsed from these tokens, in the module given in the first argument and with the variables declarations specified in the second. In case of a parsing error, this function is reduced to the term `noStratParse(`n`)` with n indicating the position of that error in the token list. When there is an ambiguity, the term `ambiguity(`$\overline{\alpha}$`,` $\overline{\beta}$`)` is obtained with two possible parses for the input tokens.

```
Maude> red metaParseStrategy(upModule('15PUZZLE, false), 'R:Row,
                             'left '; 'match '0 'R) .
rewrites: 2
result Strategy: 'left[none]{empty} ; match '__['0.Zero,'R:Row] s.t. nil
```

```
Maude> red metaParseStrategy(upModule('15PUZZLE, false), 'R:Row,
                             'luft '| 'rigth) .
rewrites: 2
result Strategy?: noStratParse(0)
```

The converse translation is supported by the `metaPrettyPrintStrategy` function, which converts a strategy metarepresentation to a list of tokens according to some printing options of sort `PrintOption` shared with `metaPrettyPrint` and described in the Maude manual [CDE+20].

```
Maude> red metaPrettyPrintStrategy(upModule('15PUZZLE, false), 'R:Row,
  ('right[none]{empty}) * ; match '__['L:Row, 's_['0.Zero], 'R:Row] s.t. nil,
  mixfix number) .
rewrites: 2
result NeQidList: 'right '* '; 'match 'L:Row '1 'R
```

Strings can be tokenized and tokens stringified using the `tokenize` and `printTokens` functions of the `LEXICAL` module in the Maude's prelude. These features facilitate the reflective manipulation of strategies and allow extending the strategy language as discussed in Chapter 7.

3.4 Set-theoretic semantics

The behavior of the strategy language expressions has been described in Section 3.1 by the results they produce for any given initial term. In conference papers prior to this thesis [EMM+07, MMV09], this description has been partially formalized as a *set-theoretic semantics* that denote a strategy expression α as a function from terms to sets of terms:

$$[\![\alpha \mathbin{@} \bullet]\!] \ : T_\Sigma \to \mathcal{P}(T_\Sigma)$$

Following this approach, we present here an updated formal semantics of the whole language. To cover all the combinators, some circumstances should be taken into account that complicate the semantic description as follows:

- Strategy definitions and **`matchrew`** operators can bind variables to values that are accessible within the definition body and the substrategies, respectively. To pass these values on, variable environments are incorporated as input to the semantic function, and we write $[\![\alpha]\!](\theta, t)$ instead of $[\![\alpha \mathbin{@} t]\!]$. Variable environments θ are represented by substitutions $\mathrm{VEnv} = X \to T_\Sigma$.

- With strategy modules, the semantic value of a strategy expression does not only depend on its sole content, but also on the strategy definitions D of the module in which it is evaluated. We describe these definitions as tuples $(sl, \overline{p}, C, \delta)$, where sl is the strategy name and δ is the expression to be executed when the input parameters match the lefthand side patterns $\overline{p}$ and satisfy the condition C. We assume that they are numbered from 1 to m, but we will later see that this order is immaterial.

  From the technical point of view, the possibility of defining recursive strategies implies that the semantics cannot be provided by a series of well-founded compositional definitions for each combinator. Hence, we have to resort to more complex tools from domain theory [Abr94]. For the presentation of the language semantics, we will assume the existence of a denotation d_k for each definition such that $d_k = [\![\delta_k]\!]$, and express the meaning of a strategy call in these terms. Later, we will justify that $\Delta = (d_1, \ldots, d_m)$ can be obtained by a fixed-point calculation.

- Recursive definitions and iterations make nonterminating executions possible. The denotation should indicate whether a computation is terminating or not, and this cannot be expressed by returning only a term set. This fact is valuable even for the compositional definition of the semantics, whose conditional expression is meant to execute its negative branch only when the condition strategy does not provide any result, which must be decided in finite time. Moreover, a nonterminating strategy evaluation can still provide solutions on other rewriting paths, as the `srewrite` and `dsrewrite` command do, so any combination of a set of results and a termination status may be possible.

For those reasons, we extend the output range of the denotation by allowing a symbol $\perp$ representing nontermination to appear in the result sets, now $\mathcal{P}(T_\Sigma \cup \{\perp\})$. However, to avoid undesired ambiguities, infinite sets will be identified regardless of whether they contain $\perp$ or not, since only nonterminating executions are able to produce infinitely many results. This motivates the following definition for any set M:

$$\mathcal{P}_\perp(M) := \frac{\mathcal{P}(M \cup \{\perp\})}{\sim} \qquad A \sim B \iff A = B \vee (A \text{ is not finite and } A \oplus B = \{\perp\})$$

where $\oplus$ stands for the symmetric difference of sets $A \oplus B = A \cup B \setminus A \cap B$. Thus, $\mathcal{P}_\perp(T_\Sigma)$ will be the range of the semantic function, and we will also use $\mathcal{P}_\perp(\text{VEnv})$ for some auxiliary operations. In the following, we do not refer to the equivalence classes explicitly but to the sets themselves, taking infinite sets with $\perp$ as representatives of their classes. Hence, we write $\perp \in A$ to express that A contains the symbol $\perp$ or it is infinite.

According to the previous comments, the final form of the denotation for a strategy $\alpha \in \text{Strat}$ is

$$[\![\alpha]\!]_\Delta : \text{VEnv} \times T_\Sigma \to \mathcal{P}_\perp(T_\Sigma)$$

As for a standard denotational definition in the domain theory framework, we have to see the class of denotations $\text{SFun} = \text{VEnv} \times T_\Sigma \to \mathcal{P}_\perp(T_\Sigma)$ as a *chain-complete partially ordered set* (ccpo), and prove that for any α the $\text{SFun}^m \to \text{SFun}$ functional that maps Δ to $[\![\alpha]\!]_\Delta$ is monotonic and continuous to calculate the definitions' semantics using the Kleene fixed-point theorem. The first step is endowing $\mathcal{P}_\perp(M)$ with an order to make it a ccpo:[1]

$$A \leq B \iff A = B \vee (\perp \in A \wedge A \setminus \{\perp\} \subseteq B)$$

Intuitively, the order expresses how results can be extended by further computation. When approaching $[\![\alpha]\!](\theta, t)$ by the results A_n of executions with at most n nested recursive calls, $\perp \in A_n$ if some recursive calls have not reached their base cases yet. These results can grow with more solutions as larger depths are allowed, and they can eventually get rid of $\perp$ or keep it forever if the execution does not terminate. On the contrary, sets without $\perp$ are definitive solutions, since all base cases have been reached, and so we call those sets *final*.

Proposition 3.1. *$(\mathcal{P}_\perp(M), \leq)$ is a chain-complete partially ordered set. Its minimum is $\{\perp\}$, and its maximal elements are M and the final sets. The union of classes is well-defined by the union of its representatives, and for any chain $F \subseteq \mathcal{P}(\mathcal{P}_\perp(M))$, $\sup F = \bigcup_{A \in F} A$ if $\perp \in A$ for all $A \in F$, and $\sup F = Z$ if there is a $Z \in F$ such that $\perp \notin Z$. In this case, Z is unique.*

The quality of being a ccpo is extended to the denotations $\text{SFun} = \text{VEnv} \times T_\Sigma \to \mathcal{P}_\perp(T_\Sigma)$ and to $\text{SFun}^m \to \text{SFun}$ by standard results.[2] Since various strategy combinators (like the concatenation) involve feeding a second strategy with the results of a first one, we will use this operation frequently. In the abstract, we can see it as a function $\text{let} : \mathcal{P}_\perp(N) \times (N \to \mathcal{P}_\perp(M)) \to \mathcal{P}_\perp(M)$ defined as follows for any $A \in \mathcal{P}_\perp(N)$ and $B : N \to \mathcal{P}_\perp(M)$:

$$\text{let}(A, B) := \{\perp : \perp \in A\} \cup \bigcup_{x \in A \setminus \{\perp\}} B(x)$$

[1]This order is a particular realization of a flat Plotkin powerdomain [Plo76].

[2]For any ccpos $(D, \leq)$ and $(E, \leq)$ and any set N, $N \to D$ is a ccpo with the order $f \leq g$ iff $f(x) \leq g(x)$ for all $x \in N$, and $E \times D$ is a ccpo with the order $(x, y) \leq (u, v)$ iff $x \leq y \wedge u \leq v$.

For readability, instead of let(A, B) we will use the informal notation let $x \leftarrow A : B(x)$ where $B(x)$ is any set expression depending on x. This functional is monotonic and continuous, and using it we can define a monotonic and continuous composition in SFun. Given two functions $f, g \in$ SFun, we define $g \circ f$ as

$$(g \circ f)(\theta, t) \coloneqq \text{let } u \leftarrow f(\theta, t) : g(\theta, u)$$

Then (SFun, $\circ$) is a monoid with identity $\mathrm{id}(\theta, t) = \{t\}$. As usual, we write f^n for the n-times composition $f \circ \cdots \circ f$ of f and f^0 for the identity.

Another prerequisite of the semantic infrastructure is matching. We assume there is a function match $: T_\Sigma(X) \times T_\Sigma(X) \to \mathcal{P}(\mathrm{VEnv})$ that provides all the minimal matching substitutions of a pattern p into a term t, i.e. all $\sigma : X \to T_\Sigma(X)$ that satisfy $\sigma(p) = t$ and $\sigma(y) = y$ for any variable $y \in X$ that does not occur in p. For matching anywhere and with extension, the functions amatch and xmatch are also considered, respectively. Their output is represented by a pair VEnv $\times\, T_\Sigma(X \cup \ominus)$ where the first component is a substitution σ, and the second is the context c where the match occurs, marked by a distinct variable $\ominus \notin X$ such that $c[\ominus / \sigma(p)] = t$. However, we often write $c(t)$ for $c[\ominus / t]$.

3.4.1 Idle and fail

The semantics of the `idle` and `fail` constants follow directly from their descriptions.

$$[\![\texttt{idle}]\!]_\Delta(\theta, t) = \{t\} \qquad [\![\texttt{fail}]\!]_\Delta(\theta, t) = \emptyset$$

3.4.2 Rule application

The evaluation of a rule application expression requires finding all matches of each rule with the given label and instantiated with the given initial substitution. In case of rules with rewriting conditions $l_i \Rightarrow r_i$, the given strategies must be evaluated in l_i, instantiated by the substitutions derived from the previous fragments, and their results matched with r_i to check the conditions and instantiate their variables. We formally describe the application of all rules labeled rl with initial substitution ρ and the mentioned strategies as

$$\begin{aligned}&\mathrm{ruleApply}(rl, \rho, \alpha_1 \cdots \alpha_m, \theta, t) = \\ &\qquad \bigcup_{\substack{(rl,l,r,C) \in R \\ \mathrm{nrewf}(C) = m}} \; \bigcup_{(\sigma_0, c) \in \mathrm{amatch}(\rho(l), t)} \text{let } \sigma \leftarrow \mathrm{check}(C, \sigma_0 \circ \rho, \alpha_1 \cdots \alpha_m, \theta) : \{c(\sigma(r))\}\end{aligned}$$

where nrewf(C) is the number of rewriting condition fragments in C. Then, the semantics of the application combinator is

$$\begin{aligned}&[\![rl\texttt{[}x_1 \texttt{ <- } t_1\texttt{, ..., } x_n \texttt{ <- } t_n\texttt{]\{}\alpha_1\texttt{, ..., }\alpha_m\texttt{\}}]\!]_\Delta(\theta, t) = \\ &\qquad \mathrm{ruleApply}(rl, \mathrm{id}[x_1 \mapsto \theta(t_1), \ldots, x_n \mapsto \theta(t_n)], \alpha_1 \cdots \alpha_m, \theta, t)\end{aligned}$$

In the definition of ruleApply, the strategies in the expression are passed to the check function. Since the strategies should be evaluated in the strategy variable context, the context substitution is also passed to the check function. Its full recursive definition is as follows:

$$\begin{aligned}\mathrm{check}(\texttt{true}, \sigma, \overline{\alpha}, \theta) &= \{\sigma\} \\ \mathrm{check}(l = r \wedge C, \sigma, \overline{\alpha}, \theta) &= \mathrm{check}(C, \sigma, \overline{\alpha}, \theta) \text{ if } \sigma(l) = \sigma(r) \text{ else } \emptyset \\ \mathrm{check}(t : s \wedge C, \sigma, \overline{\alpha}, \theta) &= \mathrm{check}(C, \sigma, \overline{\alpha}, \theta) \text{ if } \sigma(t) \in T_{\Sigma/E,s}(X) \text{ else } \emptyset \\ \mathrm{check}(l := r \wedge C, \sigma, \overline{\alpha}, \theta) &= \textstyle\bigcup_{\sigma' \in \mathrm{match}(\sigma(l), \sigma(r))} \mathrm{check}(C, \sigma' \circ \sigma, \overline{\alpha}, \theta) \\ \mathrm{check}(l \Rightarrow r \wedge C, \sigma, \alpha\overline{\alpha}, \theta) &= \text{let } t \leftarrow [\![\alpha]\!]_\Delta(\theta, \sigma(l)) : \textstyle\bigcup_{\sigma' \in \mathrm{match}(\sigma(r), t)} \mathrm{check}(C, \sigma' \circ \sigma, \overline{\alpha}, \theta)\end{aligned}$$

The range of the check function is $\mathcal{P}_\bot(\mathrm{VEnv})$ since the evaluation of rewriting condition fragments may not terminate. However, if the rule does not contain any such fragment, the result is always a plain finite substitution set.

If the rule application is surrounded by the `top` modifier, the matching must only occur on top, so the amatch in the ruleApply definition is replaced by a match. Finally, the `all` strategy constant represents a standard rewrite step with the rules in the current module. The rule's rewriting fragments are resolved by an unrestricted search, which is recursively equivalent to using `all` $*$ as the controlling strategy of the fragment.

3.4.3 Tests

Tests evaluate to a singleton set with the initial term whenever there is a match of the pattern such that the condition is satisfied. When the pattern does not match the term or no match makes the condition hold, its value is the empty set.

$$[\![\texttt{match}\ P\ \texttt{s.t.}\ C]\!]_\Delta(\theta, t) = \begin{cases} \{t\} & \exists \sigma \in \mathrm{match}(\theta(P), t) \quad \mathrm{check}(C, \sigma \circ \theta) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

The matching pattern is previously instantiated by the context substitution. Notice that the last two parameters of check have been omitted since C is an equational condition. In the following, we write $\mathrm{mcheck}(P, t, C, \theta)$ for the union of $\mathrm{check}(C, \sigma \circ \theta)$ for every $\sigma \in \mathrm{match}(\theta(P), t)$. Other test variants, such as `xmatch` and `amatch`, use their corresponding matching functions instead.

3.4.4 Regular expressions

The semantic value of concatenation is the composition of denotations $[\![\alpha\,;\beta]\!]_\Delta = [\![\beta]\!]_\Delta \circ [\![\alpha]\!]_\Delta$. This means that its results are the collection of the results of β for each result of α. In case the whole calculation of α does not terminate, the $\bot$ symbol is propagated to the whole. The alternation operator $\alpha \,|\, \beta$ has a straightforward definition

$$[\![\alpha \,|\, \beta]\!]_\Delta(\theta, t) = [\![\alpha]\!]_\Delta(\theta, t) \cup [\![\beta]\!]_\Delta(\theta, t)$$

as well as the iteration strategy

$$[\![\alpha*]\!]_\Delta(\theta, t) = \bigcup_{n \geq 0} [\![\alpha]\!]^n_\Delta(\theta, t) \ \cup\ \{\bot \ :\ [\![\alpha]\!]^n_\Delta(\theta, t) \neq \emptyset \text{ for all } n \in \mathbb{N}\}$$

The second clause of the definition makes the iteration nonterminating when its body α can be executed indefinitely, even if the terms produced by the iteration are finitely many. The alternative, considering that the evaluation of the strategy terminates even in the presence of cyclic iterations, is perfectly reasonable but we find it less appropriate. While the only direct difference between the two alternatives is the inclusion of $\bot$ in the denotation, this may produce greater consequences when combined with the conditional operator explained in the next subsection, where the termination of the condition matters.

From these definitions and $\alpha\,+ \equiv \alpha\ ;\ \alpha\,*$, it follows that $[\![\alpha+]\!]_\Delta(\theta, t) = \bigcup_{n \geq 1} [\![\alpha]\!]^n_\Delta(\theta, t)$.

3.4.5 Conditionals

As described in Section 3.1, the condition of the *if-then-else* operator is a strategy itself that is evaluated to decide which branch to take. Any result for the condition α is continued by the positive branch, and discards the execution of the negative one.

$$[\![\alpha\ ?\ \beta : \gamma]\!]_\Delta(\theta, t) = \begin{cases} [\![\beta]\!]_\Delta \circ [\![\alpha]\!]_\Delta(\theta, t) & \text{if } [\![\alpha]\!]_\Delta(\theta, t) \neq \emptyset \\ [\![\gamma]\!]_\Delta(\theta, t) & \text{if } [\![\alpha]\!]_\Delta(\theta, t) = \emptyset \end{cases}$$

The negative branch γ is only executed if the evaluation of α terminates without obtaining any solution. When $[\![\alpha]\!]_\Delta(\theta,t) = \{\bot\}$, neither β nor γ are evaluated, and the result of the conditional is $\{\bot\}$ by the first case.

The semantics of the derived operators can be obtained from the previous definitions. For example,

$$[\![\alpha \;\texttt{or-else}\; \beta]\!]_\Delta(\theta,t) = \begin{cases} [\![\alpha]\!]_\Delta(\theta,t) & [\![\alpha]\!]_\Delta(\theta,t) \neq \varnothing \\ [\![\beta]\!]_\Delta(\theta,t) & \text{otherwise} \end{cases}$$

$$[\![\texttt{test(}\alpha\texttt{)}]\!]_\Delta(\theta,t) = \begin{cases} \{t\} & \bot \notin [\![\alpha]\!]_\Delta(\theta,t) \neq \varnothing \\ \{\bot\} & \bot \in [\![\alpha]\!]_\Delta(\theta,t) \\ \varnothing & \text{otherwise} \end{cases}$$

3.4.6 Rewriting of subterms

The rewriting of subterms operator is easily defined compositionally as

$$\begin{aligned}&[\![\texttt{matchrew}\; P \;\texttt{s.t.}\; C \;\texttt{by}\; x_1 \;\texttt{using}\; \alpha_1\texttt{,}\ldots\texttt{,}\; x_n \;\texttt{using}\; \alpha_n]\!]_\Delta(\theta,t)\\ &= \bigcup_{\sigma \in \mathrm{mcheck}(t,P,C,\theta)} \text{let } t_1 \leftarrow [\![\alpha_1]\!]_\Delta(\sigma,\sigma(x_1)),\ldots,t_n \leftarrow [\![\alpha_n]\!]_\Delta(\sigma,\sigma(x_n)) : \{\sigma[x_1 \mapsto t_1,\ldots,x_n \mapsto t_n](P)\}\end{aligned}$$

where $\mathrm{mcheck}(t,P,C,\theta) \coloneqq \bigcup\{\mathrm{check}(C,\sigma_m \circ \theta) : \sigma_m \in \mathrm{match}(\theta(P),t)\}$. The matched subterms $\sigma(x_k)$ are rewritten using the strategy α_k in the variable context $\sigma \circ \theta$, and their results replace them in the subject term, by reinstantiating the pattern with the modified matching substitution. Each combination of subterm results generates a potentially different solution, and if any of the subterm computations contains $\bot$, so does the matchrew.

The variations of the matchrew combinator, `amatchrew` and `xmatchrew`, have similar definitions but replacing match by the appropriate matching function, and rebuilding the matching context.

3.4.7 Pruning of alternative solutions

This semantics does not take the `one` operator into account, because that would add another layer of nondeterminism and complicate the understanding of this exposition. Although the rewriting process controlled by strategies can be nondeterministic, the set of results shown by the `srewrite` command and described in this section are ideally deterministic in the absence of `one`. However, `one(`α`)` nondeterministically chooses one of the results that α produces, passing from the set $[\![\alpha]\!]_\Delta(\theta,t)$ to any singleton set $\{u\}$ with u in the previous set. In that case, the denotations should produce elements of $\mathcal{P}(\mathcal{P}_\bot(T_\Sigma))$.

3.4.8 Strategy calls

Strategy calls are resolved using the definition context $\Delta = (d_1,\ldots,d_m)$,

$$[\![sl(t_1,\ldots,t_n)]\!]_\Delta(\theta,t) = \bigcup_{(sl,p_1\cdots p_n,C,\delta_k)\in D} \; \bigcup_{\sigma \in \mathrm{mmatch}(\theta(t_1)\cdots\theta(t_n),p_1\cdots p_n,C)} d_k(\sigma,t)$$

where $\mathrm{mmatch}(t_1 \cdots t_n, p_1 \cdots p_n, C)$ is defined as the union of $\mathrm{check}(C,\sigma)$ for all minimal substitutions satisfying $\sigma(p_k) = t_k$ for all k. In other words, all strategy definitions matching the input arguments and satisfying the condition are executed, and their results are gathered in the final one. If no definition can be activated, the result is thus the empty set.

3.4.9 Well-definedness and strategy definition calculation

The definition of the semantics for any environment $\Delta = (d_1, \dots, d_m)$ is exhaustive and well-founded. However, it still remains pending how to formally construct such an environment where $d_k = [\![\delta_k]\!]_\Delta$. As anticipated in the introduction, the solution relies on the Klenee's fixed-point theorem and the monotonicity and continuity of the denotations.

Theorem 3.1. *For any strategy expression α, $[\![\alpha]\!]_{(d_1,\dots,d_m)}$ is monotone and continuous in $d_1, \dots, d_m$, and so is the operator $F : \mathrm{SFun}^m \to \mathrm{SFun}^m$*

$$F(d_1, \dots, d_m) \coloneqq \left([\![\delta_1]\!]_{(d_1,\dots,d_m)}, \dots, [\![\delta_m]\!]_{(d_1,\dots,d_m)} \right)$$

Hence, F has a least fixed point $\mathsf{FIX}\, F \in \mathrm{SFun}^m$ which can be calculated as

$$\mathsf{FIX}\, F = \sup \{ F^n(\{\bot\}, \dots, \{\bot\}) : n \in \mathbb{N} \}$$

Then, the denotation for the definition k is formally defined as

$$d_k \coloneqq (\mathsf{FIX}\, F)_k \, ,$$

and satisfies $d_k = F_k\,(\mathsf{FIX} F) = [\![\delta_k]\!]_\Delta$.

If Maude were running in an unbounded-memory machine, the strategy search command `srewrite in SM :` t `using` α would eventually return any solution in $[\![\alpha]\!]_\Delta(\mathrm{id}, t)$, and it would finish if and only if $\bot \notin [\![\alpha]\!]_\Delta(\mathrm{id}, t)$. In the same situation, `dsrewrite` will also return the full set of solutions, but if $\bot$ is in the denotation, some solutions may be missed.

3.5 Small-step operational semantics

The denotational semantics detailed in the previous section characterizes the results of complete strategy computations. However, we are also interested in the intermediate states, in the rewriting paths that are allowed or discarded by this controlled rewriting process. Moreover, strategies may also describe nonterminating executions that can be useful for the specification of continuously-running models like reactive systems, and these are not captured by the previous semantics. Operational semantics are a suitable tool to fill this gap. The first prototype implementation of the strategy language in Maude is actually an executable rewriting-based operational semantics [MMV09]. However, it is hard to follow the rewriting paths using it, since its states may aggregate multiple subject terms at different points of the strategic program. Hence, we proposed a nondeterministic small-step operational semantics for the strategy language [RMP⁺19a], which will be the basis for the application of model checking on strategy-controlled Maude specifications, in Chapter 4.

The execution of a strategy at any given moment depends both on the subject term being rewritten and on the execution state of the strategic program, so the configurations or execution states of the operational semantics should combine both. The simplest state is a pair $t \,@\, \alpha$ consisting of a term and a strategy expression to be evaluated in it, but some combinators and strategy calls require extending the second argument into a stack $t \,@\, \alpha_1 \cdots \alpha_n$ of pending strategies and active strategy calls. Moreover, the parallel rewriting of subterms and the evaluation of rewriting conditions need states with a nested structure.

Definition 3.1 (execution state). *A* context stack *is a word in* Stack = (Strat ∪ VEnv)*. *An* execution state $q \in \mathcal{XS}$ *is:*

1. *A pair $t \,@\, s$ composed of a term $t \in T_\Sigma(X)$ and a context stack $s \in$ Stack.*
2. *An expression of the form*

$$\mathrm{subterm}(x_1 : q_1, \dots, x_n : q_n; t) \,@\, s$$

where $x_1, \dots, x_n \in X$, $q_1, \dots, q_n \in \mathcal{XS}$, $t \in T_\Sigma(X)$ and $s \in$ Stack.

3. *An expression of the form*
$$\mathrm{rewc}(p : q, \sigma, C, \overline{\alpha}, \theta_s, r, c; t) \mathbin{@} s$$
where $p, t, r \in T_\Sigma(X)$*,* $q \in \mathcal{XS}$*,* $\overline{\alpha} \in \mathrm{Strat}^*$*, a context* c*,* $\sigma, \theta_s \in \mathrm{VEnv}$*,* C *a rule condition, and* $s \in \mathrm{Stack}$.

Execution states are univocally associated to the subject term they are currently rewriting, although it may not always be explicitly visible. The projection from execution states to terms is called $\mathrm{cterm} : \mathcal{XS} \to T_\Sigma$ and it is defined as follows.

1. $\mathrm{cterm}(t \mathbin{@} s) = t$
2. $\mathrm{cterm}(\mathrm{subterm}(x_1 : q_1, \ldots, x_n : q_n; t) \mathbin{@} s) = t[x_1 / \mathrm{cterm}(q_1), \ldots, x_n / \mathrm{cterm}(q_n)]$
3. $\mathrm{cterm}(\mathrm{rewc}(p : q, \sigma, C, \overline{\alpha}, \theta_s, r, c; t) \mathbin{@} s) = t$

The precise meaning of these states and definitions will be clarified when they are used in the following subsections. Moreover, context stacks are associated an active variable context, which will usually correspond to the *local variables* of the last strategy call. The function $\mathrm{vctx} : \mathrm{Stack} \to \mathrm{VEnv}$ is defined recursively,

$$\mathrm{vctx}(\theta s) = \theta \qquad \mathrm{vctx}(\alpha s) = \mathrm{vctx}(s) \qquad \mathrm{vctx}(\varepsilon) = \mathrm{id}$$

where id is the identity function. States with an empty stack $t \mathbin{@} \varepsilon$ are called solutions.

The small-step semantics will be given by a collection of rules defining two different relations: system steps $\to_s$ and control steps $\to_c$. The first are actual rule rewrites on the underlying rewriting system, while the second do auxiliary work to advance the strategy execution. We write $\to_{s,c} \;\coloneqq\; \to_s \cup \to_c$ for the union of both, and $\twoheadrightarrow \;\coloneqq\; \to_c^* \circ \to_s$ for a single system step preceded by some preparatory control transitions. This latter relation will be used to define the rewriting paths described by a strategy, since it enjoys the property that its steps $q \twoheadrightarrow q'$ correspond to one-step rule rewrites $\mathrm{cterm}(q) \to_R^1 \mathrm{cterm}(q')$ in the underlying rewriting system.

Every combinator is given meaning by one or more rules in the following subsections. Section 3.5.9 concludes the presentation of the small-step semantics making the rewriting paths described by strategy expressions explicit. However, if we only look at the solutions reachable with these relations, the new semantics is equivalent to that of the previous section.

Theorem 3.2. *For all* $t, t' \in T_\Sigma$*, strategy expression* α*, and* $\theta \in \mathrm{VEnv}$*,*

$$t' \in [\![\alpha]\!](\theta, t) \iff t \mathbin{@} \alpha\, \theta \to_{s,c}^* t' \mathbin{@} \varepsilon$$

and $\bot \in [\![\alpha]\!](\theta, t)$ *iff there is an infinite derivation from* $t \mathbin{@} \alpha\, \theta$*.*

3.5.1 Idle and fail

The `idle` strategy is a no-op, it does nothing on the subject term, so its control rule simply pops it from the stack.

$$t \mathbin{@} \texttt{idle}\; s \to_c t \mathbin{@} s$$

The rest of the stack is symbolically represented in the metavariable s. There is no rule for `fail` since a failure stops the execution of the strategy on this path.

3.5.2 Rule application

Rule application is easy when there are no rewriting conditions. It can be defined by a rule that replaces the current term with the rewritten one as defined by the ruleApply function in Section 3.4. The active variable context vctx(s) is written θ for brevity.

$$t \mathbin{@} rl[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]\; s \to_s t' \mathbin{@} s \quad \text{if } t' \in \mathrm{ruleApply}(rl, \mathrm{id}[x_k \mapsto \theta(t_k)]_{k=1}^n, \varepsilon, \theta, t)$$

This transition is a system step $\rightarrow_s$ because it applies a rewrite rule on the subject term.

Applying rules with rewriting conditions is more complex, since strategies must be evaluated in a subsearch for each rewriting fragment. This subsearch is supported by the rewc execution state, which also holds the substitution σ from the matching, the remaining condition C or C', the strategies α_k for the next rewriting condition fragments, the variable context θ to determine the value of the variables in α_k, the righthand side r of the selected rule, the context c defining the subterm where the rule has been applied, and the initial term t.

$$\frac{\begin{array}{c}(rl, l, r, C) \in R, \mathrm{nrewf}(C) = m, C = C_0 \wedge l_1 \Rightarrow r_1 \wedge C', \\ (\sigma, c) \in \mathrm{amcheck}(l, t, C_0, \mathrm{id}[x_1 \mapsto \theta(t_1), \ldots, x_n \mapsto \theta(t_n)])\end{array}}{\begin{array}{l}t \,@\, rl[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]\{\alpha_1, \cdots, \alpha_m\}\, s \\ \qquad \rightarrow_c \mathrm{rewc}(r_1 : \sigma(l_1) \,@\, \alpha_1 \theta, \sigma, C', \alpha_2 \cdots \alpha_k, \theta, r, c; t) \,@\, s\end{array}}$$

This control rule tries to match any rule with as many rewriting fragments as strategies are given between curly brackets in the subject term, evaluating only the equational condition C_0 that precedes the first rewriting fragment $l_1 \Rightarrow r_1$. The subsearch for solutions of the rewriting fragment $\sigma(l_1) \,@\, \alpha_1 \theta$ is advanced by the rules of the semantics as usual. Nevertheless, both system and control transitions on the substate are seen as control transitions on the whole state, since they are not rewriting the subject term but an auxiliary one.

$$[\mathrm{rewc}]\ \frac{q \rightarrow_{s,c} q'}{\mathrm{rewc}(p : q, \sigma, C, \overline{\alpha}, \theta, r, c; t) \,@\, s \rightarrow_c \mathrm{rewc}(p : q', \sigma, C, \overline{\alpha}, \theta, r, c; t) \,@\, s}$$

When a solution of the substate is found and it matches the righthand side of the fragment p, the substitution σ is extended to σ' and the rest of the condition is handled. If there are more rewriting fragments, the rewc configuration is updated to evaluate it.

$$\frac{\sigma' \in \mathrm{mcheck}(p, t', C_0, \sigma)}{\begin{array}{c}\mathrm{rewc}(p : t' \,@\, \varepsilon, \sigma, C_0 \wedge l \Rightarrow p' \wedge C, \alpha\overline{\alpha}, \theta, r, c; t) \,@\, s \\ \rightarrow_c \mathrm{rewc}(p' : \sigma'(l) \,@\, \alpha\, \theta, \sigma', C, \overline{\alpha}, \theta, r, c; t) \,@\, s\end{array}}$$

Otherwise, once the last rewriting fragment is solved, the subject term can be finally replaced by the righthand side r instantiated by the accumulated substitution σ, extended to σ' by the potentially remaining equational condition C_0.

$$\mathrm{rewc}(p : t' \,@\, \varepsilon, \sigma, C_0, \overline{\alpha}, r, c; t) \,@\, s \rightarrow_s c(\sigma'(r)) \,@\, s \quad \text{if } \sigma' \in \mathrm{mcheck}(p, t', C_0, \sigma)$$

This last step is the only system step of the sequence, since only at this point the subject term is replaced by the result of the rule application.

3.5.3 Tests

A test is popped from the stack when its pattern matches the subject term and the condition holds. Otherwise, the execution is stuck.

$$t \,@\, (\texttt{match}\ P\ \texttt{s.t.}\ C)\, s \rightarrow_c t \,@\, s \quad \text{if } \mathrm{mcheck}(P, t, C, \theta) \neq \emptyset$$

As mentioned before, tests behave like `idle` when they are satisfied and as a `fail` when they are not.

3.5.4 Regular expressions

Simple control rules handle the basic control combinators. The concatenation $\alpha\,;\beta$ removes itself from the stack, and then pushes β and α, so that β is executed when the evaluation of α has finished.

$$t \,@\, (\alpha\,;\beta)\, s \rightarrow_c t \,@\, \alpha\, \beta\, s$$

The disjunction $\alpha \,|\, \beta$ nondeterministically continues the execution with one of α and β.

$$t \,@\, (\alpha \,|\, \beta)\; s \to_c t \,@\, \alpha\; s \qquad\qquad t \,@\, (\alpha \,|\, \beta)\; s \to_c t \,@\, \beta\; s$$

Finally, the iteration strategy $\alpha*$ also offers two alternative sequels, either to break from the loop or to start a new execution of its body α.

$$t \,@\, (\alpha*)\; s \to_c t \,@\, s \qquad\qquad t \,@\, (\alpha*)\; s \to_c t \,@\, \alpha\; (\alpha*)\; s$$

3.5.5 Conditionals

As explained in previous sections, the conditional strategy $\alpha \mathbin{?} \beta : \gamma$ behaves like $\alpha \,;\, \beta$ when α does not fail. This analogy gives the rule for the positive branch.

$$t \,@\, \alpha \mathbin{?} \beta : \gamma\; s \to_c t \,@\, \alpha\; \beta\; s$$

This rule is unconditional and valid even if α does not produce any result, since β is never evaluated in that case, as the execution would have run aground before.

The rule for the negative branch, the *else* rule, is probably the most delicate rule of the semantics, and it explicitly requires the failure of the condition as premise.

$$[\text{else}]\ \frac{\text{all } \to_{s,c}\text{-executions from } t \,@\, \alpha \text{ are finite and do not contain solutions}}{t \,@\, (\alpha \mathbin{?} \beta : \gamma)\; s \to_c t \,@\, \gamma\; s}$$

The requirement is that no solution can be found on the evaluation of the condition $t \,@\, \alpha\ \mathrm{vctx}(s)$, yet we also demand that all executions from that state are finite. Note that this condition is not an orthodox premise for a structural operational semantics, since it is not a small-step of certain substate of the target deduction, but a second-order undecidable formula. With the finiteness constraint, we ensure that no creative ways of deciding the satisfaction of the premise are used to trigger the rule when the blind exhaustive execution of the semantics on the condition state is not enough. However, this is implicitly required by the deduction procedure for the standard structural operational semantics, which consists of constructing a finite proof tree from the inference rules, which can be easily extended to support this heterodox rule by requiring a finite tree of finite proof trees as a proof for its premise. Another more standard way of expressing this is possible with another auxiliary execution state and an explicit search similar to that of the rewc state. However, we prefer the clear high-level meaning of this rule to the alternative.

In practice, the conditional can be efficiently evaluated considering both rules as part of a whole. In case the execution of the $t \,@\, \alpha\beta\; s$ configuration arrives to evaluate the positive branch β, the condition is satisfied and the negative branch should not be executed. Otherwise, if all the executions from that initial state have failed before reducing α, the premise of the else rule is satisfied and so the negative branch can be triggered.

3.5.6 Rewriting of subterms

The operational semantics of the rewriting of subterms operator is defined using a structured subterm state holding an execution state for each subterm being rewritten.

$$\frac{\sigma \in \mathrm{mcheck}(P, t, C, \theta)}{\begin{array}{c} t \,@\, \texttt{matchrew}\ P\ \texttt{s.t.}\ C\ \texttt{by}\ x_1\ \texttt{using}\ \alpha_1, \ldots, x_n\ \texttt{using}\ \alpha_n\ s \\ \to_c \mathrm{subterm}(x_1 : \sigma(x_1) \,@\, \alpha_1\, \sigma, \ldots, x_n : \sigma(x_n) \,@\, \alpha_n\, \sigma ; \sigma_{-\{x_1,\ldots,x_n\}}(P)) \,@\, s \end{array}}$$

There are as many successors for the `matchrew` combinator as matches of the pattern in the subject term that satisfy the condition, with substates $\sigma(x_k) \,@\, \alpha_k\, \sigma$ for each matched subterm $\sigma(x_k)$. The substitution below α_k is σ since these strategies may contain variables both from the outer context and from the pattern and condition. In order to reconstruct the decomposed term later, the pattern P is stored in the state with all variables except those being rewritten instantiated by the matching substitution.

The parallel, or more precisely concurrent, rewriting of the subterms is achieved by applying the rules of the semantics on the substates in any order and without restrictions.

$$\frac{q_k \to_\bullet q'_k}{\text{subterm}(\dots, x_k : q_k, \dots; t) \,@\, s \to_\bullet \text{subterm}(\dots, x_k : q'_k, \dots; t) \,@\, s}$$

The $\bullet$ symbol can be either s or c, so that system (control) steps in a substate are system (control) steps on the whole state. In fact, a system step from q_k to q'_k is a one-step rule rewrite from cterm(q_k) to cterm(q'_k), and according to the definition of cterm, this is one-step rule rewrite in a subterm of the projection of the subterm state,

$$t[\dots, x_k \,/\, \text{cterm}(q_k), \dots] \to^1_R t[\dots, x_k \,/\, \text{cterm}(q'_k), \dots],$$

and so is a one-step rewrite on the whole projection by definition of the rewriting relation.

When all the substates are solutions, the original term is reassembled with the selected subterms replaced by the rewritten ones.

$$\text{subterm}(x_1 : t_1 \,@\, \varepsilon, \dots, x_n : t_n \,@\, \varepsilon; t) \,@\, s \to_c t[x_1 \,/\, t_1, \dots, x_n \,/\, t_n] \,@\, s$$

This rule and the first one should be defined similarly for the `amatchrew` and `xmatchrew` variants, where mcheck is replaced by the appropriate function and term contexts are taken into account.

3.5.7 Pruning of alternative solutions

In this operational semantics, as in the denotational one, the `one` combinator is not considered. The style and purpose of this semantics, where nondeterministic choices evolve independently, is not suitable for pruning alternative solutions at this level. However, this can be easily achieved at its metalevel, by searching the semantic graph appropriately.

3.5.8 Strategy calls

Strategy calls are resolved using the definitions D in the current module.

$$\frac{(sl, p_1 \cdots p_n, C, \delta) \in D \;\wedge\; \sigma \in \text{mmatch}(\theta(t_1) \cdots \theta(t_n), p_1 \cdots p_n, C)}{t \,@\, sl\,(t_1, \dots, t_n)\, s \to_c t \,@\, \delta\, \sigma\, s}$$

The lefthand side of every definition for the strategy *sl* is matched against the call expression evaluated in the active context and its possible conditions are checked. This is expressed here by the mmatch function defined in Section 3.4.8. Then, the call strategy is nondeterministically replaced by the righthand side expression δ of any matching definition, whose matching substitution σ is pushed before as a new variable context to give value to the definition variables and delimit the strategy call frame. In order to return from strategy calls and pop their variable contexts σ, another rule is added to the semantics.

$$t \,@\, \sigma\, s \to_c t \,@\, s$$

A strategy call executed as the last action before returning from a previous call receives the name of *tail call*, and tail calls are usually optimized in functional programming languages. The reason is that the context of the callee needs not be preserved anymore and can be replaced by that of the caller. For strategy calls, instead of pushing the context for the caller on top of the substitution of the callee, we can directly replace this substitution. Semantically, this optimization is partially irrelevant, because the possible outcomes from $t \,@\, \sigma' \sigma s$ and $t \,@\, \sigma' s$

are the same. However, the number of states reachable from a strategy call are potentially reduced, to the point where nonterminating tail-recursive strategy calls may yield a finite number of states if their executions are cyclic. Changing the finiteness of the reachable states has implications in the semantics due to the rule for the negative branch of the conditional, so we should always specify whether the optimization is being considered and in which conditions. For instance, the strategy execution engine behind the `srewrite` and `dsrewrite` commands applies the optimization only for parameterless tail-recursive calls, while the model checker for strategy-controlled systems applies it for all of them.

3.5.9 Abstract strategies and semantic traces

Using the small-step operational semantics, we can easily denote an expression in the strategy language as a more abstract extensional or intensional strategy, which are explained in Section 2.2. Remember that $\mathcal{XS}$ stands for the set of all execution states, and cterm $: \mathcal{XS} \to T_\Sigma$ for the projection of the current term for a state. A quick review of the previous rules shows that control transitions $\to_c$ do not alter the current term of a state, while $\to_s$ transitions correspond to one-step rule rewrites. Hence, the derived relation $\twoheadrightarrow$ also applies a single rule rewrite, as anticipated at the beginning of the section. Consider the sets of complete finite and infinite executions of a strategy α using $\twoheadrightarrow$ from an initial term t, where $\twoheadrightarrow^a$ is a $\twoheadrightarrow$ step whose final system transition $\to_s$ applies a rule with label a,

$$\begin{aligned}\mathrm{Ex}^*(\alpha, t) &:= \{q_0 a_1 q_1 \cdots a_n q_n \;:\; q_0 = t \,@\, \alpha, q_k \twoheadrightarrow^{a_{k+1}} q_{k+1}, q_n \to_c^* t' \,@\, \varepsilon, t' \in T_\Sigma(X)\} \\ \mathrm{Ex}^\omega(\alpha, t) &:= \{q_0 (a_k q_k)_{k=1}^\infty \;:\; q_0 = t \,@\, \alpha, q_k \twoheadrightarrow^{a_{k+1}} q_{k+1}\}\end{aligned}$$

Infinite traces have been considered without any restriction, but only finite executions ending in a state where a solution can be reached by control steps have been included in $\mathrm{Ex}^*(\alpha, t)$. The aim is to exclude incomplete execution traces, where the evaluation of part of the strategy is still pending, and namely those failed attempts that cannot be continued in any way. In particular, those semantic states that do not lead to a solution or to an infinite execution are discarded as if they have never been visited. We will call all other states valid, as defined by the following predicate.

$$\mathrm{valid}(q) := q \to_c^* \mathrm{cterm}(q) \,@\, \varepsilon \;\vee\; (\exists\, (q_n)_{n=0}^\infty \quad q \twoheadrightarrow q_0 \;\wedge\; \forall k \in \mathbb{N} \quad q_k \twoheadrightarrow q_{k+1})$$

The execution traces of the semantics are actually rewriting paths of the underlying rewriting system, if we forget about the strategy continuation and concentrate on the terms they hold, i.e. if they are projected by the cterm function. As usual, this function can be extended to $(\mathcal{XS} \cup A)$-words leaving actions unchanged with $\mathrm{cterm}(q_0 a_1 q_1 \cdots) = \mathrm{cterm}(q_0)\, a_1\, \mathrm{cterm}(q_1) \cdots$, and to sets by $f(A) = \{f(a) \;:\; a \in A\}$. With this useful notation, the extensional strategy denoted by α is the following.

Definition 3.2. *Given a strategy expression α and a term $t \in T_\Sigma$,*

$$E(\alpha) := \bigcup_{t \in T_\Sigma} E(\alpha, t) \quad \textit{where } E(\alpha, t) := \mathrm{cterm}(\mathrm{Ex}^*(\alpha, t)) \cup \mathrm{cterm}(\mathrm{Ex}^\omega(\alpha, t))$$

The extensional strategy $E(\alpha)$ is closed, as follows from the intensional definition of the Ex^ω set. Thus, it can be expressed as an intensional strategy, although losing the information about the finite executions.

$$\begin{aligned}\lambda_\alpha(t a_1 t_1 \cdots a_n t_n) := \{(a, t') \in A \times T_\Sigma \;:\; & t \,@\, \alpha \twoheadrightarrow^{a_1} q_1 \twoheadrightarrow^{a_2} \cdots \twoheadrightarrow^{a_n} q_n \twoheadrightarrow^{a} q', \\ & \mathrm{cterm}(q_k) = t_k, \mathrm{valid}(q')\}\end{aligned}$$

Sometimes we do not care about the transition labels and remove them from these definitions, so that extensional strategies are T_Σ^∞ words instead of $(T_\Sigma \times A)^\infty$ elements. More properties of the strategy $E(\alpha)$ as a language are discussed in the following section.

3.6 Expressiveness of the language

Using the small-step operational semantics of the previous section, a strategy expression α can be described as a set of rewriting paths $E(\alpha) \subseteq \Gamma_{(T_\Sigma, \to_R^1)} \subseteq T_\Sigma^\infty$, i.e. as a sublanguage of the language of all executions of the rewrite theory $\mathcal{R} = (\Sigma, E, R)$. From this perspective, the expressive power of the Maude strategy language can be naturally studied. For example, we can say that the language is Turing complete, in the sense that it has the ability to denote any recursively enumerable subset of rewriting paths. This property is trivially met since the strategy language includes stateful recursive definitions, in whose arguments we can even simulate a Turing machine. In the following, we assume that the languages under consideration only contain finitely many different symbols in T_Σ, i.e. that their alphabets are finite. Abusing notation, we write $\Gamma_\mathcal{R}$ for $\Gamma_{(T_\Sigma, \to_R^1)}$.

Proposition 3.2. *For any ∞-recursively enumerable language $L \subseteq \Gamma_\mathcal{R}$, there is some strategy expression α such that $E(\alpha) = L$.*

However, the strategy language is rarely used as a theoretical tool to describe complex languages, but as a pragmatic resource to solve specific problems where the evaluation of the strategy is expected to finish. When the evaluation of a strategy from an initial term terminates, its denotation is in the more restrictive class of ∞-regular languages, and the converse is almost true. We say that a strategy terminates in a term if the reachable states from $t \,@\, \alpha$ are finitely many.

Definition 3.3. *The set of* reachable states *from a state $q \in \mathcal{XS}$ is $\{q' \in \mathcal{XS} : q \to_{s,c}^* q'\}$. The set of reachable terms from q is $\bigcup_{q' \in R} \mathrm{terms}(q')$ where R are the reachable states from q and*

$$\mathrm{terms}(q) = \mathrm{cterm}(q) \cup \begin{cases} \mathrm{terms}(q_1) \cup \cdots \cup \mathrm{terms}(q_n) & \textit{if } q = \mathrm{subterm}(x_1 : q_1, \ldots, x_n : q_n; t) \\ \mathrm{terms}(q') & \textit{if } q = \mathrm{rewc}(x : q', \ldots) \\ \{\mathrm{vctx}(s)(t_1), \ldots, \mathrm{vctx}(s)(t_n)\} & \textit{if } q = t \,@\, sl\,(t_1, \ldots, t_n)\, s \end{cases}$$

The fact that the reachable states from q are finitely many does not necessarily mean that all executions from q are terminating, since they could loop within that finite set of states. This is meaningless when finding the results of a strategy with the `srewrite` command, but very useful when model checking strategy-controlled systems, as we will see in Chapter 4. The following proposition formalizes what we have claimed in the previous paragraph.

Proposition 3.3. *If the reachable states from $t \,@\, \alpha$ are finitely many, $E(\alpha, t)$ is a closed ∞-regular language.*

The converse of this proposition is not true, since the strategy expression `empty(0)` and the definition `empty(N) := fail | empty(s(N))` with `N` of sort `Nat` clearly show. The strategy `empty` does not apply any rewrite, and so the language denoted by `empty(0)` is empty, which is ∞-regular. However, infinitely many execution states of the form $t \,@\, \texttt{fail}\, [\texttt{N} \mapsto n]$ for $n \in \mathbb{N}$ can be reached (or $t \,@\, \texttt{fail}\, [\texttt{N} \mapsto n][\texttt{N} \mapsto n-1] \cdots [\texttt{N} \mapsto 0]$) without the tail-call optimization). The problem is obviously the nonterminating and superfluous recursive call, and we can find an alternative strategy expression for the same language with a finite execution span, namely `fail`. This possibility is general.

Proposition 3.4. *If L is a closed ∞-regular language, there is a strategy expression β such that $E(\beta) = L$ and the reachable states from $t \,@\, \beta$ are finitely many for all $t \in T_\Sigma$.*

Since the strategy language includes the constructors of regular expressions, this property should not seem surprising. However, the Kleene star is not faithfully represented by the iteration, and so the premise requires that the language L is closed. For example, consider a rewriting system with two terms `a` and `b`, and two rules $X \to^{\texttt{id}} X$ and $\texttt{a} \to^{\texttt{ab}} \texttt{b}$. The language

$E = \{\mathtt{a}\}^+\{\mathtt{b}\}^\omega$ is ω-regular and an extensional strategy in that system. It is not closed, because it lacks its limit $\mathtt{a}^\omega$, and so no expression with a finite set of reachable states can represent it according to Proposition 3.3. In particular, the natural candidate `id * ; ab ; id-omega` where `id-omega := id ; id-omega` denotes the closure $\overline{E}$ of the E strategy where $\mathtt{a}^\omega$ is included. This does not contradict Proposition 3.2 because there is an expression whose denotation is exactly E: `count(0)` using the following strategy definitions.

```
strats count prefix : Nat @ Letter .
sd count(N) := prefix(N) | count(s(N)) .
sd prefix(0) := ab ; id-omega .
sd prefix(s(N)) := id ; prefix(N) .
```

However, the execution of `count(0)` yields infinitely many execution states. At the end of this section, we will come back to the iteration and to how the closed restriction can be removed.

The previous propositions relate the language properties of strategies with the finiteness of their reachable execution state sets. Relating this with syntactical properties of the strategy expressions will also be interesting and useful. Strategies are potentially complex recursive programs that depend on the rewriting system and equational theory in which they are applied, so simple conditions can only be obtained for very particular cases. In general, strategies without recursive calls or iterations always produce finitely many states, but their usefulness is very limited. Assuming that only a finite number of terms are visited by the strategy, iterations can be safely used. Under the same conditions, tail-recursive strategies may also keep the state space finite if the tail call optimization is enabled and they are not called at run time with infinitely many distinct arguments, even if they do not terminate. We consider that a strategy is recursive if it is inside a loop in the static call graph, i.e. the graph that links the strategy names with their definitions and these definitions with the strategy names that appear in their strategy calls. A strategy call is a tail call if it is the last action before returning from a previous call, but syntactically we find them just before the end of strategy definitions.

Definition 3.4. *All recursive calls in a strategy expression are tail if the expression is:*

- **`idle`**, **`fail`**, *a test, or a strategy call expression.*
- $\alpha \,|\, \beta$ *and all recursive calls in* α *and* β *are tail.*
- $\alpha \,;\, \beta$ *and* α *does not contain recursive calls and all recursive calls in* β *are tail.*
- $\alpha \mathrel{?} \beta : \gamma$ *and* α *does not contain recursive calls, and all recursive calls in* β *and* γ *are tail.*
- *A subterm rewriting or rule application expression, and all recursive calls in its substrategies are tail.*

Proposition 3.5. *The reachable states from* $t \mathbin{@} \alpha$ *are finitely many if any of the following conditions holds:*

1. α *does not contain iterations or recursive calls.*
2. *The reachable terms from* $t \mathbin{@} \alpha$ *are finitely many and all recursive calls in* α *and the reachable strategy definitions are tail.*

The assumptions in the second case of this proposition are not only static properties of the strategy expression, but also dynamic properties of its execution. This would be difficult to check in some cases, but it will be easy for many usual strategies. For example, the reachable states from the expression `id * ; ab ; id-omega` that appeared some paragraphs above are finitely many, since only two states can be visited, `a` and `b`, the only recursive call `id-omega` is tail recursive and it does not take arguments.

Proposition 3.5 is purely qualitative and does not address how much the number of reachable states can grow compared to the number of reachable or total terms of the original rewriting system. Finding a tight bound looking only at the strategy expression is impossible, because the number of execution states essentially depends on the underlying system. In fact, strategies are often used to explore a subset of the state space of the uncontrolled model, and so it will be awkward to bound the number of reachable states by the product of the (possibly infinite) size of this state space and the number of possible strategy continuations. However, we provide the following proposition with a loose bound that could be useful in some situations.

Proposition 3.6. *Given a strategy expression α without strategy calls*

$$|\{\textit{reachable states from } t \mathbin{@} \alpha\}| \leq |\{\textit{reachable terms from } t \mathbin{@} \alpha\}|\; N(\alpha)$$

where

- $N(\texttt{idle}) = 2$
- $N(\texttt{fail}) = 1$
- $N(\texttt{match}\, P \;\texttt{s.t.}\; C) = 2$
- $N(\alpha \,|\, \beta) = N(\alpha) + N(\beta) + 1$
- $N(\alpha\,;\beta) = N(\alpha) + N(\beta) + 1$
- $N(\alpha*) = N(\alpha) + 2$
- $N(\,\alpha \;?\; \beta : \gamma) = N(\alpha) + N(\beta) + N(\gamma) + 2$
- $N(rl\,[\rho]\{\alpha_1, \ldots, \alpha_n\}) = \sum_{k=1}^{n} N(\alpha_k) + 2$
- $N(\texttt{matchrew}\; P \;\texttt{s.t}\; C \;\texttt{by}\; x_1 \;\texttt{using}\; \alpha_1,\; \ldots, x_n \;\texttt{using}\; \alpha_n) = \prod_{k=1}^{n} N(\alpha_k) + 2$

Moreover, terminating calls $sl\,(t_1, \ldots, t_n)$ can be replaced by its unrolling paying 2 for each replacement or 3 if it receives arguments. If `st` *is a tail-recursive strategy without arguments defined by α, the number of reachable states can be bounded with the formula above assuming $N(\texttt{st}) = 1$.*

3.6.1 The iteration and the Kleene star

The Kleene star usually designates all the finite concatenations of words in a given language, and this is the meaning it has in ω-regular expressions. However, in the Maude strategy language, the strategy α^* admits both the finite consecutive iterations of α and its nonterminating iteration, which can be informally described as $\alpha^* \mid \alpha^\omega$. When discussing Proposition 3.3 we have seen an example of this. In effect, the following steps of that example

$$\texttt{a}\mathbin{@}(\texttt{id}*)\;\texttt{ab}\,;\,\texttt{id-omega} \to_c \texttt{a}\mathbin{@}\texttt{id}\;(\texttt{id}*)\;\texttt{ab}\,;\,\texttt{id-omega} \to_s^{\texttt{id}} \texttt{a}\mathbin{@}(\texttt{id}*)\;\texttt{ab}\,;\,\texttt{id-omega}$$

can be repeated indefinitely in a loop according to the semantics. This inaccuracy is completely irrelevant when executing a strategy to obtain its results, but it may be interesting in some cases for model checking. Making the iteration behave like the Kleene star would allow capturing fairness restrictions of the model in the strategy itself. In order to update the previous semantics to faithfully represent the Kleene star, we only need to remove those executions of a strategy where an iteration is repeated infinitely many times in a row.

Definition 3.5. *Given $\pi \in \mathcal{XS}^\omega$, we say that π iterates forever if any of the following conditions hold:*

1. *$\pi_k = t_k \mathbin{@} c_k\, \alpha*\, s$ for all $k \in \mathbb{N}$, and there are infinitely many $k \in \mathbb{N}$ such that $\pi_k = t_k \mathbin{@} \alpha*\, s$ and $\pi_{k+1} = t_k \mathbin{@} \alpha\, \alpha*\, s$.*
2. *$\pi_k = \mathrm{subterm}(x_1 : \rho_{1,k}, \ldots, x_n : \rho_{n,k}; t)$ for all $k \in \mathbb{N}$, and ρ_m iterates forever for some $1 \leq m \leq n$.*
3. *π^k iterates forever for some $k \in \mathbb{N}$.*

An execution π in $(\mathcal{XS}, \to_{s,c})$ iterates finitely if it does not iterate forever, and an execution π in $(\mathcal{XS}, \twoheadrightarrow)$ iterates finitely if its expansion to $\to_{s,c}$ transitions iterates finitely.

Intuitively, an execution iterates forever if the transition $t \,@\, \alpha^*\, s \rightarrow_c t \,@\, \alpha\, \alpha^*\, s$ is repeated infinitely many times for the same iteration, inside a subterm state or at the top level. Definition 3.6 introduces the extensional strategy $E_K(\alpha) \subseteq E(\alpha)$ for a strategy expression α that respects the semantics of the Kleene star.

Definition 3.6. *Given a strategy expression α,*

$$E_K(\alpha) \coloneqq \bigcup_{t \in T_\Sigma} E_K(\alpha, t) \quad \textit{where } E_K(\alpha, t) \coloneqq \mathrm{cterm}(\mathrm{Ex}^*(\alpha, t)) \cup \mathrm{cterm}(\mathrm{Ex}_K^\omega(\alpha, t))$$

where $\mathrm{Ex}_K^\omega(\alpha, t) \coloneqq \{\pi \in \mathrm{Ex}^\omega(\alpha, t) \,:\, \pi \textit{ iterates finitely}\}$.

The set $E_K(\alpha)$ is not necessarily closed, and so it could not be represented by an intensional strategy λ. In the example, $E_K($`id *; ab ; id-omega`$) = \{$`a`$\}^*\{$`b`$\}^\omega$, because the execution that appears at the beginning of this subsection has been purged. In general, this semantic refinement requires removing the *closed* adjective of Proposition 3.3, and allows dropping it from the premise of Proposition 3.8, so that any ∞-regular language can be denoted with a finite execution space. Actually, the acceptance condition in the definition of Ex_K^ω can be rephrased as a Streett and sometimes co-Büchi acceptance condition, as we will see in Section 4.5.

Proposition 3.7. *If the reachable states from $t \,@\, \alpha$ are finitely many, $E_K(\alpha, t)$ is an ∞-regular language.*

Proposition 3.8. *If L is an ∞-regular language, there is a strategy expression β such that $E_K(\beta) = L$ and the reachable states from $t \,@\, \beta$ are finitely many for all $t \in T_\Sigma$.*

In Section 4.5, we also discuss how strategies interpreted under this semantics can express model-checkable fairness conditions.

3.7 Implementation

Like the strategy language design, its implementation was started long before this thesis. The first prototype was written by Alberto Verdejo in Maude as an extension of Full Maude [MMV04]. Most of the current language is supported by that prototype, although its strategy modules are not part of the module hierarchy, and they cannot include other modules or take them as parameters. After some experimentation with this prototype, an efficient integration of the strategy language within the Core Maude interpreter was started by Steven Eker [EMM+07]. At the beginning of this thesis, this implementation included the infrastructure of processes or tasks that allows the execution of strategies, most strategy combinators were implemented, and the `srewrite` command was available to execute them. However, the development of the language was in an impasse due to other priorities for Maude. Moreover, it was not documented in the Maude manual, and the different examples using the strategy language published during that time were based on the previous prototype. The most significant missing features were strategy modules and recursive strategies, the **`matchrew`** combinator for rewriting subterms, and the **`one`** operator. These and other unplanned features, like the `dsrewrite` command for a depth-first search of strategy solutions, have been added in the first part of this work.

The interpreter of Maude, sometimes called Core Maude in contrast with Full Maude, is implemented in C++. Its implementation is organized in different modules or source directories, like `Utility` for some convenient data structures and functions, `Core` for basic objects that are independent of the specific equational theories, `Parser` for the lexical analyzer, `Mixfix` for the syntactical analyzer and most of the interpreter implementation, `Meta` for reflective stuff, and `Interface` for the common interface of terms and associated entities that are specialized for the different equational theories like `FreeTheory`, `S_Theory`, `ACU_Theory`, etc.[3] There is

[3]The overall structure of the implementation of the first version of Maude is described in [CDE+02]. These general ideas are still valid, although Maude has evolved since then and some information is outdated or incomplete.

also a module for the `StrategyLanguage`, where strategy expressions are represented and their execution is implemented by means of a collection of processes and tasks. They are depicted in Figure 3.4. However, other important components of the strategy language are included in different modules.

Strategy expressions are represented and parsed in a straightforward manner as trees of linked instances of the `StrategyExpression` subclasses. Tests, rule applications, strategy calls, and other combinators include terms or refer to elements of the module where they are evaluated, so strategy expressions are logically associated to a fixed module. The execution of strategies is more subtle as already realized in [EMM+07]. Strategy evaluation commands should explore the rewriting graph permitted by the strategy, but this graph may contain loops and other nonterminating rewriting sequences. Moreover, recursion and the existence of rewriting conditions in rules allow a state to have infinitely many successors. Consequently, the goal of the search algorithm, on a both potential infinitely branching and infinitely deep tree, is to support fairness: whenever t can be rewritten to t' using a strategy, the solution t' should be found in a finite computation. Nevertheless, this computation may not be practical in terms of memory and time.

The execution engine of strategies consists of a series of *processes* and *tasks*, which are shown in Figure 3.4 next to the combinators to which they are related. Processes make the tree grow by processing expressions, applying rules, and calling strategies. Tasks are used to confine subsearches and continuations, and delimit variable contexts. Looking at the initial Maude prototype [MMV09], processes can be understood as its `<_@_>` tasks, and tasks as `<_;_>` terms. The tasks, processes, and expressions implemented during this thesis are those for calls, for the rewriting of subterms, and for the **`one`** operator.

3.7.1 Processes

Processes are the active components of the execution infrastructure, which are executed in a round-robin policy to achieve the fair execution of `srewrite`, or in a FIFO policy for the `dsrewrite` command. Each process is associated to a term and to a position in the stack of pending strategies. Terms are indexed in a hash table that is maintained along the execution of a strategy command. The stack of pending strategies is also an indexed global structure that simultaneously holds the stacks of all active and inactive processes. In an overall view, the structure is actually a forest where stacks are completely or partially shared by different processes.

The execution always begins with the ubiquitous *decomposition process*, which checks and pops the top of its stack of pending strategies, and then executes the appropriate action for the strategy expression it finds. These actions are specified in each instance of a virtual method `decompose` of the strategy expression class. For example, for **`idle`** it does nothing so that the new top of the stack is executed in the next turn of the process; for **`fail`**, it terminates the process; for a test, it acts as an **`idle`** or a **`fail`** depending on the satisfaction of the condition; for the concatenation $\alpha ; \beta$, it pushes β and then α on the stack; for the disjunction $\alpha \mid \beta$, the decomposition process is cloned, α is pushed on the clone's stack, and β on the original one; for the iteration $\alpha *$, the process in cloned similarly and $\alpha *$ followed by α is only pushed in the current one; for a conditional operator $\alpha ? \beta : \gamma$, the current process is destroyed after creating a task and a new decomposition process to evaluate the condition; and for strategy calls, rule applications, and subterm rewriting operators, one of their homonym processes is created to take on the search for their corresponding matches. When this process reaches the bottom of the stack, it informs its parent that the current term is a solution.

The process in charge of the actual rewriting is the *application process*. In each turn, it tries to make a new rewrite by finding the next position and rule that can be applied, considering the given label and initial substitution. Strategy calls behave similarly using *call process*, which find in each turn another strategy definition or matching within the same definition that can be executed. In that case, it creates a decomposition process for the definition expression,

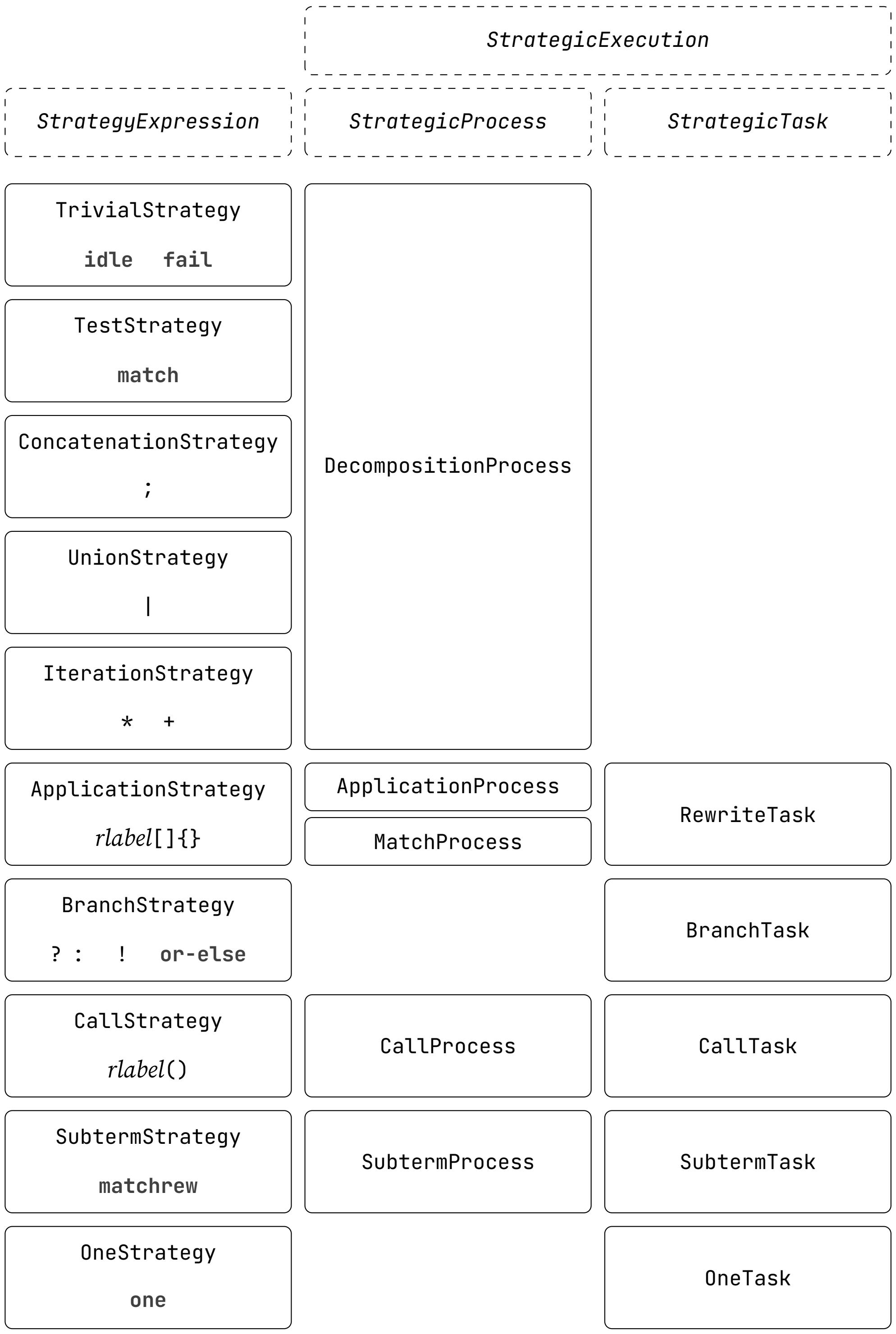


Figure 3.4: Chart of classes of the strategy language implementation.

possibly in a new *call task* that confines its variable context. As an optimization of a common case, call processes are not generated for strategies that are defined by a single unconditional statement without input parameters, whose definition expression is directly pushed on top of the stack. Further optimizations are applied to tail calls, for which a call task is not created if avoidable. Finally, *subterm processes* also follow the same principles and find all matches of the `matchrew` main pattern in the subject term satisfying the condition, and then create a decomposition process for each selected subterm with its associated strategy. All of them are controlled by a *subterm task*, which is explained in the next subsection.

3.7.2 Tasks

Tasks support subsearches and enclose variable contexts, and since these can be nested, they form a hierarchy. Each task gathers a collection of processes, and it is usually associated to a position in the stack of pending strategies that would be resumed when the subsearch yields a result. The root of the task hierarchy is a `StrategicSearch` object that holds the table of terms, the stacks of pending strategies, and the stacks of variable contexts. This object also controls the search for strategy solutions and executes the strategic processes, differently for the `srewrite` and `dsrewrite` commands in either of its two subclasses.

For example, the conditional operator α ? β : γ needs to know whether α has reached a solution to discard or activate the evaluation of the negative branch. This is done by the already-mentioned *branch task* that, when informed by one of its processes that a solution has been reached, creates a decomposition process on the parent task with the original pending stack with β pushed on top. However, if the branch task runs out of processes without finding a solution, the negative branch is executed by a decomposition process from the initial term and for γ. Their derived operators, `not`, `test`, `try`, `or-else`, and the normalization `!` one, are executed under the same task with some specific optimizations. The execution of `one(`α`)` is achieved by a task that executes α and interrupts the subcomputation as the first solution is reported.

Subsearches are also required for the evaluation of rewriting condition fragments, using *rewrite tasks* and *match processes*, and for the rewriting of subterms. In this case, a *subterm task* is created for each successful matching of the `matchrew` pattern in the subject term, as we have explained before. Each subterm is processed in a separate child task under the subterm task with the strategies specified in the `matchrew`. The subterm task keeps a table with all the results obtained from each subterm, which grows as the processes in their tasks notify new solutions. Since these belong to the same process list, subterms are rewritten concurrently. When there are solutions for every subsearch, the original subject term is reassembled with its subterms replaced by the solutions in any possible combination. These combinations are generated and notified as solutions to the parent by a private process of the subterm task, one at a time. Finally, *call tasks* house the evaluation of strategy calls and separate their variable contexts.

The avoidance of repetitive work is another important mission of tasks. Each task counts with a hash set of pairs of subject terms and stack positions already seen during the evaluation of previous processes. Decomposition processes check in their parent tasks whether they are going to decompose a strategy that has already been executed for the same term, and in that case they terminate their execution without evaluating it twice. This mechanism does not only improve the efficiency of strategic search, but it also affects its semantics by removing execution cycles in iterations that can be executed indefinitely repeating the same states. As a consequence of the optimization of tail calls, cycles are also detected in nonterminating tail-recursive strategies without arguments, so that their execution will finish if they visit a finite number of states. However, they are not detected in nonterminating tail-recursive strategies with arguments, since this does not have clear application when looking for solutions using the `(d)srewrite` commands to justify the cost of storing and comparing the variable contexts of previous calls. However, describing infinite executions is meaningful when model checking, so cycles are detected for those recursive strategies in the model checker of Chapter 4. In case the detection of repetitive work were extended to strategy calls, it may be worth not using tasks to hold strategy calls but adding the variable context as a new component of the sets of seen states.

3.7.3 Modules

Strategy modules and theories are implemented like their functional and system counterparts. Internally, both modules and theories of any kind are objects of the same subclasses of a `Module` class, where two new statements have been added as additional attributes: strategy declarations and strategy definitions. These are represented using the new classes `RewriteStrategy` and `StrategyDefinition` in the `Core` module. Strategy objects hold the signature of their declaration and manage the list of their definitions, so that the latter can be quickly checked on a strategy call. Strategy definitions associate a call term in their lefthand sides to a strategy expression in its righthand side, with a possibly empty condition. When a module is processed, variables are checked to be bound and other static restrictions are verified.

Matching strategy calls to strategy definitions uses a tricky implementation detail. Strategy call expressions are not terms on the module where they are evaluated, but a strategy expression object with a list of term arguments. Matching each argument separately is not as efficient as matching all the arguments as a whole, since the matching algorithms in Maude leverage the relations between them.[4] An auxiliary tuple symbol of a hidden sort `strategy[internal]` is defined in the module to build terms for the strategy calls and the lefthand sides of definitions, so that the efficient Maude algorithms can be properly used.

3.7.4 Metalevel

The reflection of the strategy language, strategy modules and theories at the metalevel consists of the declaration of their metarepresentation in the modules `META-STRATEGY`, `META-MODULE`, and `META-THEORY` of the Maude prelude. However, some more routine changes are needed in the `Meta` module of the C++ implementation to translate back and forth between metarepresentation and the usual representation of strategy expressions and statements. The `metaSrewrite` descent function does this conversion and then executes the strategy just like the `srewrite` and `dsrewrite` commands, depending on the `SrewriteOption` argument.

3.8 Comparison with other strategy languages

In Section 2.2.3, we have mentioned some other strategy languages, like ELAN [BKK⁺01], Stratego [BKV⁺08], Tom [BBK⁺07], and ρLog [MK06]. The design of the Maude strategy language was influenced by ELAN and Stratego [EMM⁺07], the design of Stratego was in turn influenced by ELAN, and Tom was developed by the same authors of ELAN. Consequently, all these languages have similar foundations and repertories of strategy combinators, as shown in Table 3.1. Most combinators of any of the languages are available in the others, or otherwise can be expressed with more elaborate expressions. However, we highlight the main differences regarding Maude:

1. Maude enforces a clear separation between rules and strategies, while in ELAN and Tom strategies can be used in the definition of rules and they are sometimes required to cooperate. This dependency is more explicit in ρLog, where strategies are defined using an extension of the syntax for defining rules. Stratego allows and promotes this separation, but also let strategies directly set the subject term, define new rules, and apply inline ones. The separation between rules and strategies in Maude is a conscious design decision

[4] For example, matching $f(L\,N\,R, N)$ against $f(1 \ldots 5000, 3)$ is efficient for the Maude algorithms, since they will match the simpler second argument first, which determine the other list variables L and R. However, matching each argument separately, we would have to process 5000 matches for the first argument.

Maude	ELAN	Stratego	Tom	ρLog
`idle`	`id`	`id`	`Identity`	`Id`
`fail`	`fail`	`fail`	`Fail`	
label	*label*	*label*	Inline rules	*label*
`top`	By default	By default	By default	By default
`;`	`;`	`;`	`Sequence`	∘
`|`	`dk`	`+`	`Choice`	`|`
`or-else`	`first`	`<+`	Using Java	`Fst`
`match` P	Using rules or library	`?`P	`%match`	Using rules
α `?` β `:` γ	`if` α `then` β `orelse` γ `fi`	α `<` β `+` γ	Using Java	Using rules
`*`	`iterate*`			`*`
`+`	`iterate+`			
`!`	`repeat*`	`repeat`	`Repeat`	`NF`
`one(`α`)`	`dc/first one(`α`)`	Implicit	Implicit	
`try`	`first(`α`, id)`	`try`	`Try`	`Fst[`α`, Id]`
`test`	Using rules	`where`	Using Java	Using rules
$\alpha \equiv sl(t_1, \ldots, t_n)$	`call(`α`)`	`call(`sl || t_1`, ..., `t_n`)`	α`.apply`	Using rules

Table 3.1: Strategy language syntax comparison.

to ease the understanding, analysis, and verification of the specification, respecting the *separation of concerns* principle.

2. Strategies are potentially nondeterministic in all of them. However, ELAN, ρLog, and Maude compute the whole set of results for a given term, while Stratego and Tom explore a single rewriting path by resolving the nondeterminism arbitrarily.

3. The application of a rule is the common basic element of the languages. However, all except Maude apply rules on top by default. Instead, Maude applies them on any subterm within the subject term, unless explicitly indicated by **`top`**.

4. Parameterization is available by means of modules in ELAN and Maude, or at the level of strategy definitions in ρLog and Stratego, whose approach is more succinct. In fact, strategy expressions in Stratego are usually simpler, since the variety of strategy combinators is wider, some combinators are generic, and the signatures of strategies are not declared (types are not checked).

5. Stratego, Tom, and ρLog allow programming generic traversals of terms, which need to be made explicit for each known operator in the signature in Maude. The generic possibilities of ρLog follow from the untyped Wolfram language of Mathematica in which it is based, since variables can be used not only for terms but also for function symbols and argument lists.

6. Maude and Stratego allow binding variables in strategy expressions. In Maude their scope is delimited by nested regions like the **`matchrew`** substrategies, while in Stratego explicit scopes can be declared and variables are defined from left to right within the same expression level.

7. ELAN, Stratego, and Tom include a library of reusable strategy definitions, while no such library currently exists for the Maude strategy language.

Some features, like the generic term traversals of Stratego (item 5), are not incorporated into the Maude language for the sake of simplicity. This is also the case of *congruence operators* that make each symbol f in the signature a strategy combinator such that f`(`α_1`, …, `α_n`)` applies the given strategies to the corresponding subterms. These combinators have been implemented in Maude at the metalevel using **`matchrew`** operators, which can be seen as their counterparts in this language, in Section 7.2 and in [RMP+20a]. Generic traversals have also been addressed using a similar reflective strategy transformation, but we briefly describe here how they can be written by hand. Specifically, the three primitives or one-step descent operators of Stratego are `all`, to apply a strategy to all the direct subterms of the subject term; `some`, to apply a strategy to all the children in which it does not fail, and as long as it succeeds in at least one; and `one`, to apply the strategy to the first subterm in which it succeeds from left to right. These can be defined in the Maude strategy language for each n-ary symbol f as:

```
sd st_all := matchrew f(x1, ..., xn) by x1 using α, ..., xn using α .
sd st_one := matchrew f(x1, ..., xn) by x1 using α or-else
              ... or-else
             matchrew f(x1, ..., xn) by xn using α .
```

The `some` operator can be defined as `test(st_one(α)) ; st_all(try(α))` in pseudocode according to its definition. It can also be given its own definition using a sequence of `matchrew`s that apply the strategy α on every argument, letting it fail in all but one argument each.

```
sd st_some := (matchrew f(x1, ..., xn) by x1 using α ;
               matchrew f(x1, ..., xn) by x2 using try(α), ...,
                                              xn using try(α)) or-else
               ... or-else
              matchrew f(x1, ..., xn) by xn using α .
```

These definitions could be parametric on α by using parameterized strategy modules.

Similarly, the choosing operators from ELAN have not been included in the Maude language, but almost all of them can be defined easily using Maude constructs:

$$\begin{aligned}
\texttt{first}(\alpha_1, \ldots, \alpha_n) &\equiv \alpha_1 \ \texttt{or-else} \cdots \texttt{or-else}\ \alpha_n \\
\texttt{dk}(\alpha_1, \ldots, \alpha_n) &\equiv \alpha_1 \mid \cdots \mid \alpha_n \\
\texttt{first one}(\alpha_1, \ldots, \alpha_n) &\equiv \texttt{one}(\alpha_1)\ \texttt{or-else} \cdots \texttt{or-else}\ \texttt{one}(\alpha_n) \\
\texttt{dc one}(\alpha_1, \ldots, \alpha_n) &\equiv \texttt{one}(\alpha_1 \mid \cdots \mid \alpha_n)
\end{aligned}$$

The exception is $\texttt{dc}(\alpha_1, \ldots, \alpha_n)$ that returns all the results of only one of the α_i chosen non-deterministically. With the `one` operator we are only able to take either one or all of the results of any strategy expression. In the opposite direction, there are Maude constructs that are not available in ELAN, but they can also be expressed by more involved strategy expressions, in the worst case resorting to rules.

3.8.1 Other Maude strategy languages

Some internal strategy languages have been implemented and used in Maude [CM96, PM02] before the proposal of the current stable strategy language [MMV04]. Since the latter is influenced by the previous, they are very similar.

Another similar strategy language was proposed by José Meseguer together with his Temporal Logic of Rewriting [Mes08, Mes07] to verify properties on infinite-state systems.

$$\begin{aligned}
b &::= \top \mid \bot \mid p \mid \neg b \mid b \wedge b \mid b \vee b \\
e &::= \mathit{idle} \mid \delta \mid \mathit{any} \mid e \wedge e \mid (e|e) \mid e\,;e \mid e{+} \mid e\,\mathbf{U}\,e \mid e.b
\end{aligned}$$

Most combinators of this strategy language are available in Maude's, including *idle*, *any* (`all`), disjunction, concatenation, and iteration. Atomic propositions p can be seen as particular cases of tests `match X s.t. X` $\models p$, `not` as the logical negation, and concatenation as the logical conjunction. In these terms, $e.b$ is an arbitrary strategy concatenated with a test. The action atoms δ describe a collection of rule applications cited by their labels, taking arguments to control the value of the rule's variables, and with the indication of the context where they can be applied. This can be easily represented by means of `matchrew` and rule application combinators,

$$\texttt{matchrew}\ f(\ldots,\texttt{X},\ldots)\ \texttt{by X using}\ \mathit{rlabel}[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$$

However, the $e \wedge e$ and $e \,\mathbf{U}\, e$ do not have equivalent in the language discussed in this thesis, since they are oriented to the verification of TLR* formulae. Their semantics is the following, described as a satisfaction relation $\mathcal{R}, w \vDash e$ on labeled executions $w \in (T_\Sigma \cup A)^*$,

$$\begin{aligned}
\mathcal{R}, w\,t\,w' \vDash e \wedge e' &\iff (\mathcal{R}, w\,t\,w' \vDash e \;\wedge\; \mathcal{R}, w\,t \vDash e') \vee (\mathcal{R}, w\,t \vDash e \;\wedge\; \mathcal{R}, w\,t\,w' \vDash e') \\
\mathcal{R}, w \vDash e \,\mathbf{U}\, e' &\iff \exists\, n < |w| \quad \mathcal{R}, w^n \vDash e' \;\wedge\; \forall k < n \quad \mathcal{R}, w^k \vDash e
\end{aligned}$$

The first one $e{\wedge}e'$ is the dual of the union $e \mid e'$, and it admits only the executions that are allowed by both e and e' as in a synchronous execution, although any of them is able to continue once the other has finished. The second one $e \,\mathbf{U}\, e'$ is like the *until* operator of the CTL* family, and e should repeatedly follow the rewrites of the path until e' does. The meaning of the iteration is also understood as in the Kleene star (see Section 3.6.1).

This language captures fairness properties that are not captured with the standard semantics of the builtin strategy language, but it is always executed on rewriting paths of bounded depth, where iterations cannot loop forever and the execution of the second strategy of the until operator cannot be postponed indefinitely. The intersection combinator $e \wedge e'$ is executable in practice without depth bounds, but it cannot be replaced generically by other operators of the builtin language. In Section 7.3.1, this operator is implemented in an extensible executable semantics of the Maude strategy language.

Chapter 4

Model checking strategy-controlled systems

The title of this thesis is *model checking strategy-controlled rewriting systems*, and this is what this chapter is about. First, we examine this subject in the most abstract terms of Section 2.2, in which understanding how the satisfaction of temporal properties on strategy-controlled system should be defined is straightforward. The very natural and general definitions fixed in the following section are respected in the rest of the chapter, where model checking is particularized to Maude specifications controlled by its strategy language. This particularization relies on the small-step operational semantics presented in Section 3.5, which is also used to transform the uncontrolled model into a strategy-aware structure where temporal properties can be effectively checked. This is implemented as an extension of the Maude LTL model checker in Section 4.3, whose internal Kripke structure is also made available to external model checkers in Section 4.4 to verify properties in CTL, CTL*, and μ-calculus. These connections are centralized in a unified interface `umaudemc` that can be easily extended with support for more logics and tools. Section 4.5 discusses model checking against the more expressive semantics of the strategy language in Section 3.6.1 whose iteration coincides with the Kleene star, for which the standard procedure cannot be directly applied. We explain how it can be used to express fairness constraints in the strategy itself and check properties under them using a prototype implementation. Further details on the implementation of the different model checkers are given in Section 4.7. Finally, Section 4.8 discusses how this topic has been covered in the literature.

Most of this chapter is based on the conference papers [RMP+19a, RMP+20c] and the journal papers [RMP+21b, RMP+21a]. However, the Kleene-star model checker and the probabilistic model-checking features appear for the first time here. Application examples and an evaluation of the model checker's performance are available in Part III.

4.1 An abstract definition

Understanding the satisfaction of temporal properties on systems controlled by strategies is clearer when they are seen in the abstract and generic terms of Section 2.2 rather than as syntactic expressions on a strategy language. Given a Kripke structure $\mathcal{K} = (S, \to, I, AP, \ell)$ and an extensional strategy $E \subseteq \Gamma_{\mathcal{K}}$, the key intuition is that checking a temporal property in $(\mathcal{K}, E)$ should be checking that property only on the executions or behaviors selected by the strategy E. For linear-time properties, which refer to every execution of the model individually, the following definition is natural and almost inexorable.

Definition 4.1. *Given a strategy-controlled system $(\mathcal{K}, E)$ and a linear-time property φ, $(\mathcal{K}, E) \vDash \varphi$ if $\mathcal{K}, \ell(\pi) \vDash \varphi$ for all $\pi \in E$.*

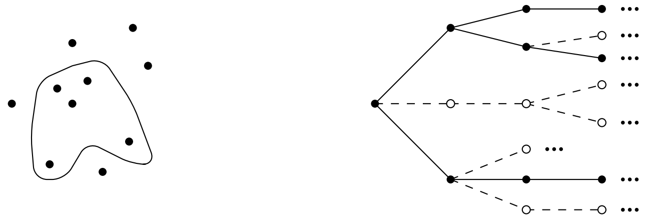


Figure 4.1: Strategies filtering executions and branches on execution trees.

Since the strategy is currently expressed as a subset E of executions of $\mathcal{K}$, this definition simply requires the satisfaction of the property on the executions contained in this subset regardless of the rest. The satisfaction relation $\vDash$ on propositional traces will always be well-defined for any temporal logic based on the atomic propositions of the model.

For branching-time properties, a similar definition can be proposed taking into account that these properties are checked on trees and strategies can be seen as subtrees of the execution trees of the original model, as explained in Section 2.2.4. Intuitively, strategies prune some branches that should not be considered when evaluating branching-time properties, as illustrated in Figure 4.1. However, definitions of branching-time logics do not usually mention trees explicitly, so we resort to an auxiliary Kripke structure to obtain a clearer definition.

Definition 4.2 (unwinding)**.** *Given a Kripke structure $\mathcal{K}$ and an intensional strategy λ, the unwinding $\mathcal{U}(\mathcal{K}, \lambda)$ of $\mathcal{K}$ according to λ is the Kripke structure $((S \cup A)^+, U, I, AP, \ell \circ \mathrm{last})$ where $(w, was) \in U$ if $(a, s) \in \lambda(w)$ and $\mathrm{last}(ws) = s$ for all $w \in (S \cup A)^*$.*

The unwinding of a transition system is a well-known concept [Ter03], but only the executions allowed by the strategy are included in this case. Seen as a graph, it is no other than the execution subtree for the strategy λ. In this definition and in the following, we refer to intensional strategies instead of extensional ones, and labeled transition systems instead of unlabeled ones. The last choice is for greater generality, since actions can be safely removed if desired. On the contrary, intensional strategies λ are less expressive than extensional ones, but being intensional is a reasonable assumption for practical strategies, as argued in Section 2.2. Non-intensional aspects can be considered later, as we do in Section 4.5. Consequently, we define the satisfaction of a branching-time property by a strategy-controlled system as its satisfaction in the unwinding:

Definition 4.3. *Let φ be a branching-time formula, $(\mathcal{K}, E(\lambda)) \vDash' \varphi$ if $\mathcal{U}(\mathcal{K}, \lambda) \vDash' \varphi$.*

This definition coincides with the previous one on linear-time properties, because the executions of the unwinding projected by last are exactly those of the strategy. For that reason, we will write $\vDash$ for both definitions in the following.

Proposition 4.1. *Given a linear-time property φ, $(\mathcal{K}, E(\lambda)) \vDash \varphi$ iff $(\mathcal{K}, E(\lambda)) \vDash' \varphi$.*

In order to apply these definitions to concrete strategy descriptions, like expressions in a strategy language, we should establish which executions are allowed by them. The nondeterministic small-step operational semantics in Section 3.5 is used for the Maude strategy language in Section 4.2, and a similar approach can be used for other languages. In any case, there are some general consequences of the previous abstract definition:

- The satisfaction of a temporal property only depends on the executions allowed by the strategy. In particular, if two concrete representations of a strategy denote the same subset of executions, they should be indistinguishable for model checking.
- Any temporal logic or property that is well-defined in the base system is also well-defined when it is controlled by a strategy.

- Conversely, the kind of properties under consideration are exactly those available for the uncontrolled systems. They do not reason about strategies, but about the model that results from their restrictions.

For linear-time properties and considering the language $L(\varphi) = \{\rho \in \mathcal{P}(AP)^\infty : \mathcal{K}, \rho \vDash \varphi\}$ of propositional traces admitted by φ or its labeled version, the model-checking problem is reduced to a language inclusion one $\ell(E) \subseteq L(\varphi)$, whose decidability, complexity, and algorithmic results can be exploited. If the property logic is LTL, $L(\varphi)$ is an ω-regular language and the problem is PSPACE-complete for any strategy E whose projection $\ell(E)$ is ω-regular, and 2EXPTIME-complete for any ω-context-free $\ell(E)$ [BEM97]. However, the program complexity (for a fixed formula) in both cases is polynomial on the size of its automaton. Moreover, if $\ell(E)$ is ω-regular the automata-theoretic approach explained in Section 2.3.2 can be applied, even if its automaton has non-trivial Büchi conditions.[1] On the contrary, if $\ell(E)$ strictly lies in higher classes of the Chomsky hierarchy, the problem is undecidable.

This connection with the language inclusion problem suggests algorithms for model checking linear-time properties on strategy-controlled systems. However, we also want to model check branching-time properties and to reuse off-the-shelf algorithms for the target logics. They cannot be directly applied on the unwinding $\mathcal{U}(\mathcal{K}, \lambda)$ as defined in Definition 4.3, because it is not finite. Nevertheless, the behavior of the strategy possibly coincides in many partial executions, so that many states of the unwinding could be bisimilar. The relation $w \sim w'$ if there is a bisimulation relating w and w' is an equivalence relation, and the quotient $\mathcal{U}(\mathcal{K}, \lambda)/\!\sim$ is surely bisimilar to the unwinding and it may be finite. Many logics are invariant by bisimulation, and so their properties can be checked equivalently on finite bisimilar Kripke structures like this quotient. The practical procedure for finding them may be specific for each strategy language or formalism. For the Maude strategy language, this finite structure will be generated if available by using the pervasive small-step operational semantics. In general, we are sure that this finite Kripke structure exists if and only if $\ell(E)$ is a closed and ω-regular language.

Proposition 4.2. *Given a Kripke structure $\mathcal{K}$ and a closed strategy $E \subseteq (S \cup A)^\omega$,*

$$(1) \Rightarrow (2) \Rightarrow (3) \Leftrightarrow (4)$$

1. *E is ω-regular.*
2. *There is a finite Kripke structure $\mathcal{K}'$ bisimilar to $\mathcal{U}(\mathcal{K}, \lambda)$.*
3. *$\ell(E)$ is ω-regular.*
4. *There is a finite Kripke structure $\mathcal{K}'$ such that $\ell'(\Gamma^\omega_{\mathcal{K}'}) = \ell(E)$.*

In case of linear-time properties, the bisimilarity requirement can be weakened to sharing only the same propositional traces, and then it is enough that $\ell(E)$ is ω-regular. Finding a bisimilar Kripke structure and applying standard algorithms on it is the program we will adopt regarding the Maude strategy language in the following sections, where the denotation of strategy expressions in Section 4.2 via the small-step operational semantics gives all the ingredients for the previous definitions. However, in Section 4.5, we will exploit the automata approach for checking systems controlled by non-closed strategies. In the following subsections, we provide straightforward generalizations to strategy-controlled systems of the textbook semantics of two temporal logics, CTL* and μ-calculus. These definitions agree and confirm the soundness of Definition 4.3, since applying them to a system $(\mathcal{K}, E(\lambda))$ is equivalent to applying the classical definitions to $\mathcal{U}(\mathcal{K}, \lambda)$.

[1] In the automata-theoretic approach (see Section 2.3), the intersection of the model automaton $L(\mathcal{K})$ and the negated property automaton $L(\neg\varphi)$ is calculated to decide $\mathcal{K} \vDash \varphi$. In this case, the automaton for $L(\mathcal{K})$ has trivial Büchi conditions and the intersection algorithm is simpler. However, if $L(\mathcal{K})$ is replaced by an $\ell(E)$ with non-trivial Büchi conditions, a similar intersection algorithm can be applied, although the required space may double [CHV+18].

4.1.1 Generalization of CTL* for strategy-controlled systems

As we have explained in Section 2.3.1, CTL* is a branching-time logic with the following grammar:

$$\begin{aligned}\Phi &::= \bot \mid \top \mid p \mid \neg\,\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathbf{A}\,\phi \mid \mathbf{E}\,\phi \\ \phi &::= \Phi \mid \neg\,\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \bigcirc\,\phi \mid \Diamond\,\phi \mid \Box\,\phi \mid \phi\,\mathbf{U}\,\phi\end{aligned}$$

Terms built from ϕ are called *path formulae* and describe properties of fixed execution paths, while terms under the Φ symbol are called *state formulae* and refer to a state of the transition system.

The semantics of CTL* is usually expressed by a satisfaction relation on states $\mathcal{K}, s \vDash \Phi$ and on paths $\mathcal{K}, \pi \vDash \phi$. However, when $\mathcal{K}$ is controlled by a strategy, a state formula like $\mathbf{E}\,\phi$ should not quantify over all paths, but only over those allowed by the strategy. Moreover, this subset of paths may depend not only on the last state but on the whole history of the execution. Consequently, the satisfaction relation for strategy-controlled systems replaces the state s by a (partially consumed) extensional strategy $\mathcal{K}, E \vDash \Phi$, and path formulae also carry a strategy in addition to the chosen path $\mathcal{K}, E, \pi \vDash \phi$ where $\pi \in E$. To carry on this information in the following recursive definition, we introduce the operation $E \upharpoonright ws := \{s\pi : ws\pi \in E\}$ that gives the execution paths allowed by a strategy $E \subseteq S^\infty$ to continue from ws. For readability, the initial $\mathcal{K}$ is omitted.

1. $E \vDash p$ iff $\forall\, \pi \in E \quad p \in \ell(\pi_0)$
2. $E \vDash \neg\,\Phi$ iff $E \nvDash \Phi$
3. $E \vDash \Phi_1 \wedge \Phi_2$ iff $E \vDash \Phi_1$ and $E \vDash \Phi_2$
4. $E \vDash \mathbf{E}\,\phi$ iff $\exists\, \pi \in E \quad (E \upharpoonright \pi_0), \pi \vDash \phi$
5. $E, \pi \vDash \Phi$ iff $E \vDash \Phi$
6. $E, \pi \vDash \neg\,\phi$ iff $E, \pi \nvDash \phi$
7. $E, \pi \vDash \phi_1 \wedge \phi_2$ iff $E, \pi \vDash \phi_1$ and $E, \pi \vDash \phi_2$
8. $E, \pi \vDash \bigcirc\,\varphi$ iff $(E \upharpoonright \pi_0\pi_1), \pi^1 \vDash \varphi$
9. $E, \pi \vDash \phi_1\,\mathbf{U}\,\phi_2$ iff $\exists\, n \geq 0 \;\; E \upharpoonright \pi^{\leq n}, \pi^n \vDash \phi_2 \,\wedge\, \forall\, 0 \leq k < n \;\; E \upharpoonright \pi^{\leq k}, \pi^k \vDash \phi_1$

Other derived operators are defined by the usual equivalences. This is a direct generalization of the classical semantic definition in Section 2.3.1, and similar variations have appeared in the literature when studying CTL* in the context of tree languages [Tho89] and other extensions of this logic. The only substantial changes are in (4), where only executions in E are considered, and in (8) and (9), where the strategy argument is updated to the allowed executions from the time point where the recursive relation is evaluated. In fact, this definition coincides with the classical relation if we take $E = \Gamma^\omega_{\mathcal{K},s}$.

Proposition 4.3. *Given a CTL* formula Φ, $\mathcal{K}, s \vDash \Phi$ iff $\Gamma^\omega_{\mathcal{K},s} \vDash \Phi$.*

Moreover, checking a CTL* property according to this generalized definition in $\mathcal{K}$ is the same as doing so with the standard definition on the unwinding or a bisimilar structure, because CTL* is invariant by bisimulation.

Proposition 4.4. *Given $(\mathcal{K}, E(\lambda))$ and a CTL* formula φ, $\mathcal{U}(\mathcal{K}, \lambda) \vDash \varphi$ iff $\mathcal{K}, E(\lambda) \vDash \varphi$.*

4.1.2 Generalization of the μ-calculus for strategy-controlled systems

Modal μ-calculus [Koz83] is an extension of Hennessy-Milner logic [HM80] with least μ and greatest ν fixed-point operators, as we have explained in Section 2.3.3.

$$\varphi \ ::= \ \bot \mid \top \mid p \mid Z \mid \neg\,\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid [a]\,\varphi \mid \langle a \rangle\,\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

The following generalization mimics the original definition in that section, but the denotation of a formula is now a set of trees or strategies $\langle\!\langle \varphi \rangle\!\rangle_\xi \subseteq \mathcal{P}(\Gamma_\mathcal{K})$ instead of a set of states. The idea is that a system controlled by strategies $(\mathcal{K}, E)$ satisfies a μ-calculus formula φ iff $E \in \langle\!\langle \varphi \rangle\!\rangle_\xi$. A valuation is now $\xi : \mathrm{Var} \to \mathcal{P}(\mathcal{P}(\Gamma_\mathcal{K}))$ and $\xi[Z/U]$ is the function ξ with its value for the variable Z replaced by U.

1. $\langle\!\langle p \rangle\!\rangle_\xi \quad = \{T \subseteq \Gamma_\mathcal{K} : \forall \pi \in T \quad p \in \ell(\pi_0)\}$
2. $\langle\!\langle \neg\varphi \rangle\!\rangle_\xi \quad = \mathcal{P}(\Gamma_\mathcal{K}) \setminus \langle\!\langle \varphi \rangle\!\rangle_\xi$
3. $\langle\!\langle \varphi_1 \wedge \varphi_2 \rangle\!\rangle_\xi \quad = \langle\!\langle \varphi_1 \rangle\!\rangle_\xi \cap \langle\!\langle \varphi_2 \rangle\!\rangle_\xi$
4. $\langle\!\langle Z \rangle\!\rangle_\xi \quad = \xi(Z)$
5. $\langle\!\langle \langle a \rangle\,\varphi \rangle\!\rangle_\xi \quad = \{T \subseteq \Gamma_\mathcal{K} : \exists\, s a \pi \in T \quad T \upharpoonright s a \pi_0 \in \langle\!\langle \varphi \rangle\!\rangle_\xi\}$
6. $\langle\!\langle \nu Z.\varphi \rangle\!\rangle_\xi \quad = \bigcup \{F \subseteq \mathcal{P}(\Gamma_\mathcal{K}) : F \subseteq \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]}\}$

For instance, the denotation of an atomic proposition p takes all strategies whose paths satisfy p in their initial terms, instead of all states that satisfy p in the classical definition. Similarly, the modality $\langle a \rangle\,\varphi$ takes all strategies with a path that satisfy φ after an a transition. As usual, for the fixpoint in (6) to be well-defined, φ must be monotone, so every variable must be under an even number of negations. The missing two operators are defined by the usual equivalences, for example $[a]\,\varphi \equiv \neg\langle a \rangle\,\neg\varphi$.

The following two results are the counterparts of Propositions 4.3 and 4.4 for CTL*, and say that the definition is actually a generalization of the classical one, and that it is coherent with the procedure proposed for model checking strategy-controlled systems.

Proposition 4.5. *Given $(\mathcal{K}, E)$ and a closed μ-calculus formula φ, $s \in [\![\varphi]\!]_{\mathcal{K},\eta}$ iff $\Gamma_{\mathcal{K},s} \in \langle\!\langle \varphi \rangle\!\rangle_{\mathcal{K},\xi}$ for any η and ξ.*

Proposition 4.6. *Given $(\mathcal{K}, E(\lambda))$ and a closed μ-calculus formula φ, $s \in [\![\varphi]\!]_{\mathcal{U}(\mathcal{K},\lambda),\eta}$ for $s\pi \in E$ iff $E \in \langle\!\langle \varphi \rangle\!\rangle_{\mathcal{K},\xi}$ for any η and ξ.*

4.2 Model checking strategy-controlled Maude models

After discussing the meaning of model checking for strategy-controlled systems in Section 4.1 and describing the rewriting paths allowed by an expression in the Maude strategy language in Section 3.5, the satisfaction of temporal properties in Maude specifications controlled by its strategy language is already unambiguously defined.

Suppose we are given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ specified in a Maude module M, and an additional signature Π of atomic propositions defined on the terms of $\mathcal{R}$ by some equations D using a satisfaction predicate $\_ \models \_$. The Kripke structure of the uncontrolled rewriting is defined as

$$\mathcal{M} := (T_{\Sigma/E}, \to^1_R, T_{\Sigma/E}, AP_\Pi, L_\Pi)$$

where $\to^1_R$ is the one-step rewrite relation,

$$AP_\Pi := \{\ \theta(p(x_1, \ldots, x_n)) \mid p \in \Pi, \theta \text{ ground substitution }\}$$

is the set of ground instances of the atomic proposition terms, and $L_\Pi([t]) := \{\theta(p(x_1, \ldots, x_n)) \in AP_\Pi \mid (E \cup D) \vdash t \models \theta(p(x_1, \ldots, x_n)) = \mathtt{true}\}$ is the labeling function that evaluates them under the equations E and D. Suppose we are also given a strategy module M' and a strategy expression α in M', possibly referring to some strategy definitions in the module, and a linear-time property φ on the previous atomic propositions; then, φ is satisfied in M' controlled by α if

$$(\mathcal{M}, E(\alpha)) \vDash \varphi \iff \mathcal{U}(\mathcal{K}, \lambda_{E(\alpha)}) \vDash \varphi \qquad \iff \forall \pi \in E(\alpha) \quad \mathcal{M}, L_\Pi(\pi) \vDash \varphi$$

according to Definitions 3.2 and 4.1, although the rightmost equivalence is only valid for linear-time properties. While the extensional strategy $E(\alpha)$ may in principle contain finite traces where logics like LTL are not properly defined, these can be extended to infinite ones by the typical stuttering extension explained in Section 2.2.

In order to reuse existing model-checking algorithms for the desired logic, the strategy suggested in Section 4.1 will be followed. For the moment, we will focus on linear-time properties and find a Kripke structure whose propositional traces coincide with $L_\Pi(E(\alpha))$. A reasonable candidate is the graph of the nondeterministic small-step operational semantics of Section 3.5,[2]

$$\mathcal{O}^{\alpha,t} := (\mathcal{XS}, \twoheadrightarrow, \{t \mathbin{@} \alpha\}, AP_\Pi, L_\Pi \circ \mathrm{cterm})$$

Indeed, the nonterminating executions of $\mathcal{O}^{\alpha,t}$ projected by the cterm function are the nonterminating rewriting paths of $E(\alpha, t)$ by definition of the latter and of $\mathrm{Ex}^\omega(\alpha, t)$. According to $\mathrm{Ex}^*(\alpha, t)$, the finite executions of the strategy α are the projections of the paths in this structure that end in states $\mathrm{Sol} := \{q \in \mathcal{XS} : q \to_c \mathrm{cterm}(q) \mathbin{@} \varepsilon\}$ where a solution can be reached by control steps. However, finite execution should be seen as infinite traces whose final state is repeated forever. Using the construction of Definition 2.3, the Kripke structure that represents the stuttering extension of $E(\alpha, t)$ can be defined as

$$\mathcal{M}_{\alpha,t} := \mathcal{O}^{\alpha,t}_{\mathrm{Sol}} = (\mathcal{XS} \times \{0\} \cup \mathrm{Sol} \times \{1\}, \twoheadrightarrow_{\mathrm{Sol}}, \{t \mathbin{@} \alpha\}, AP_\Pi, L_\Pi \circ \mathrm{cterm} \circ \pi_1)$$

where $\pi_1(q, n) = q$ is the projection on the first component. Remember that $\twoheadrightarrow_{\mathrm{Sol}}$ extends $\twoheadrightarrow$ with safe self-loops for solution states, where finite executions are allowed to terminate. In the following, the disjoint union $\mathcal{XS} \times \{0\} \cup \mathrm{Sol} \times \{1\}$ will be confused when possible with $\mathcal{XS}$.

Proposition 4.7. *The projections of the infinite traces of $\mathcal{M}_{\alpha,t}$ by* $\mathrm{cterm} \circ \pi_1$ *coincide with the stuttering extension of $E(\alpha, t)$.*

However, the abstract construction of $\mathcal{O}^{\alpha,t}_{\mathrm{Sol}}$ can be applied more efficiently in this particular case. In effect, there are three relevant situations regarding finite traces, shown in Figure 4.2. In the third case, where the solution state has a successor that allows continuing the execution, its duplication is justified. This situation may occur for example after executing β in the strategy $\beta*$, when both finishing the iteration and continuing with β are possible. If the loop were added directly to the solution state, spurious executions would be allowed that stay a number of steps in the solution state and then continue by its successor. This situation cannot happen in the second case, where the solution state does not have successors, so a loop can be safely added to it without duplication.

In the first case, the state is not a solution, but one in which the strategy has failed. Since no loop is added to it and by considering only the nonterminating executions of $\mathcal{M}_{\alpha,t}$, this execution state is completely ignored, as well as all other states from which neither solution states nor infinite executions can be reached. From the point of view of the strategy, these states and the executions that go through them have been discarded by an explicit `fail`, a failed test, an inapplicable rule, etc., and so they are seen as if they have never happened. These failed states do not disturb the standard on-the-fly LTL algorithm described in Section 2.3.2 because its nested depth-first search will not find any cycle through them. They can be removed in

[2]Assuming that the Kripke structure has a single initial state instead of finitely many is without loss of generality, since each initial state can be treated separately.

$$\begin{tikzcd} {} \arrow[r, two heads] & \texttt{fail} \end{tikzcd}$$

(1) Dead end

$$\begin{tikzcd} {} \arrow[r, two heads] & t' \mathbin{@} \varepsilon \arrow[loop right, dashed, two heads] \end{tikzcd}$$

(2) Deadlock solution state

$$\begin{tikzcd} & & q' \\ {} \arrow[r, two heads] & q \arrow[ur, two heads] \arrow[dr, dashed, two heads] & \\ & & t' \mathbin{@} \varepsilon \arrow[loop right, dashed, two heads] \end{tikzcd}$$

(3) Continuable solution state

Figure 4.2: Solution and deadlock states in $\mathcal{O}^{\alpha,t}$ and their adjustments.

linear time on the number of states by exploring the rewriting graph as in the Tarjan's strongly connected components algorithm [Tar72], but this is incompatible with on-the-fly model checking because the entire graph might need be explored to conclude that a single state is valid. However, this removal algorithm must be surely applied for other model-checking algorithms that do not enjoy this property, like tableau-based methods for LTL.

In conclusion, the rewriting system controlled by α can be checked against linear-time properties with the standard algorithms for the target logic using the just defined $\mathcal{M}_{\alpha,t}$.

Corollary 4.1. $(\mathcal{M}, E(\alpha, t)) \vDash \varphi \iff \mathcal{M}_{\alpha,t} \vDash \varphi$ *for any linear-time property* φ.

Model checking is usually decidable when the Kripke structure is finite and the transition relation and labeling function are decidable. Some conditions on the underlying rewriting system that are common with the standard model checker must be fulfilled [EMS04], but also some others related to the strategy. Given a strategy-controlled specification as explained before, the $\mathcal{M}_{\alpha,t}$ is finite and $\twoheadrightarrow_{\mathrm{Sol}}$ and L_Π decidable if:

- The set of reachable execution states from $t \mathbin{@} \alpha$ by $\to_{s,c}$ is finite (this implies $\to_{s,c}$ and $\twoheadrightarrow$ are decidable, see Lemma B.14).

- The rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by M plus the equations D defining the predicates Π and the equations G defining auxiliary operations Ω used to define the strategies in `M'` satisfy:

  - all of E, $E \cup D$, $E \cup D \cup G$ are (ground) Church-Rosser and terminating perhaps modulo axioms, where $(\Sigma, E) \subseteq (\Sigma \cup \Pi, E \cup D) \subseteq (\Sigma \cup \Pi \cup \Omega, E \cup D \cup G)$ is a protecting extension, i.e. it adds neither junk nor confusion to what it extends, and
  - R is (ground) coherent relative to E perhaps modulo axioms.

We are not saying that these are decidability conditions for model checking because no temporal logic has been fixed yet. However, these conditions are valid for all temporal logics considered in this thesis. In Section 3.6, we proved that the reachable states from any execution $t \mathbin{@} \alpha$ are finitely many almost if and only if $E(\alpha, t)$ is ω-regular. Other sufficient conditions have been found in Proposition 3.5 based on syntactical aspects of the strategy expressions, which can now be used to deduce the decidability of model checking for a concrete strategy. How strategy-controlled Maude specifications are prepared for model checking and how linear-time properties are effectively checked is discussed in Section 4.3.

Branching-time properties have not been considered in the previous discussion. However, would it be correct to apply branching-time model-checking algorithms to $\mathcal{M}_{\alpha,t}$? In other words, is this structure bisimilar to the unwinding $\mathcal{U}(\mathcal{K}, \lambda_{E(\alpha)})$? The answer is no, and one of the problems is related to the failed states that we have mentioned when reviewing self-loops, which are not admissible for branching-time algorithms in general. This can be easily solved by purging the failed states by an exploration of the state space. There is another more serious problem that we explain in the following subsection.

4.2.1 Strategies and branching-time properties in Maude

The transition system yielded by the semantics is not ready for model checking branching-time properties, as seen in the previous section. However, the main reason is that states which are logically the same in the underlying system may be seen as distinct states due to the strategy continuation they hold, changing the tree structure of the model and making it depend on syntactical aspects of the strategies. We will illustrate this problem with an example of a simple vending machine:

```
mod VENDING-MACHINE is
  sorts Soup Thing Machine .
  subsort Thing < Soup .

  ops e a c : → Thing [ctor] .
  op _[_] : Soup Soup → Machine [ctor] .

  op empty : → Soup [ctor] .
  op __ : Soup Soup → Soup [ctor asoc comm id: empty] .

  vars O I : Soup .

  rl [put1]  : O e [I]     ⇒ O   [I e] .
  rl [apple] : O   [I e]   ⇒ O a [I] .
  rl [cake]  : O   [I e e] ⇒ O c [I] .
endm
```

The vending machine is a term $O\ [I]$ where O represents the belongings of its user and I the contents of its internal coin box. The machine can receive one euro coin `e` with the rule `put1`, and sells apples `a` and cakes `c` for one and two euros, respectively. Let us consider the strategies

$$\alpha \equiv \texttt{put1 ; apple | put1 ; put1 ; cake} \quad \text{and} \quad \beta \equiv \texttt{put1 ; (apple | put1 ; cake)}.$$

These two different strategy expressions are essentially the same, because their abstract denotations coincide $E(\alpha) = E(\beta)$, so the vending machine must satisfy the same properties whether controlled by α or β according to Definition 4.3. Intuitively, these strategies can be identified with the plans of a person using the machine, where α has already decided which item to buy before inserting any coin, and β delays the choice until the first coin is inserted. An external observer looking at the user interaction with the machine will not be able to distinguish when this choice has been made, it is not part of the *observable behavior*, and so it should be irrelevant for any property considered. This principle would not be obeyed if we applied standard algorithms on $\mathcal{M}_{\alpha,t}$ as Figure 4.3 shows. There, we can see the execution trees by the $\twoheadrightarrow$ relation from an initial configuration with two coins `e e [empty]` using both α (left) and β (right). Disregarding the strategy continuations after the `@` sign, i.e. projecting the nodes by the cterm function, we obtain rewriting trees where terms are connected by one-step rule rewrites. However, the tree for α cannot be considered a subtree of the execution tree of $\mathcal{M}$ because it contains repeated children. In any case, the branching structures of the execution trees for α and β and of their projections are manifestly different, and so they can be distinguished by branching-time temporal properties, as the CTL property $\mathbf{A} \bigcirc \mathbf{E} \Diamond\, \mathit{hasCake}$ attests.

```
mod VENDING-MACHINE-PREDS is
  protecting VENDING-MACHINE .
  including SATISFACTION .

  sort Machine < State .
  op hasCake : → Prop [ctor] .
```

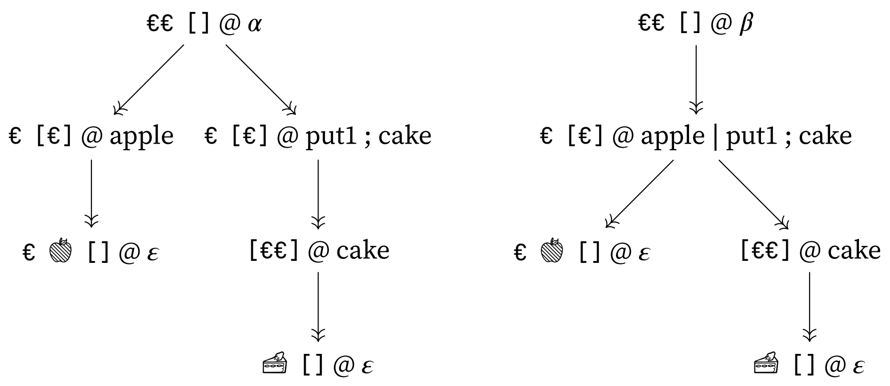


Figure 4.3: Strategy rewrite graph for α and β.

```
   vars I O : Soup .
   eq O c [I] ⊨ hasCake = true .
   eq O   [I] ⊨ hasCake = false [owise] .
 endm
```

In effect, in the immediate successors of the root of the α tree the full path is already chosen, and in the left one no cake is ever bought. On the contrary, there is only one immediate successor of the initial state for β, where we can still choose the right branch to get the cake.

The ambiguity on the satisfaction of the formula by the strategy $E(\alpha) = E(\beta)$ should be avoided. In this example, the problem would be solved if the two successors of the root in the execution tree for α were combined into a single state, whose projection will be well-defined since they share the same term. Merging successors with a common base term is a general solution to the problem that can be applied locally, solves the ambiguity, and produces a Kripke structure bisimilar to the unwinding of the strategy as desired. The following definition formalizes this construction and the removal of failed states discussed in the previous section. Remember that a state is valid

$$\text{valid}(q) \coloneqq \exists\, t \in T_\Sigma \quad q \to^*_{s,c} t \,@\, \varepsilon \quad \vee \quad \exists\, (q_n)_{n=1}^{\infty} \quad q \twoheadrightarrow q_1 \twoheadrightarrow q_2 \twoheadrightarrow \cdots$$

if a solution can be reached or a nonterminating execution can be followed from it (or both, since the two possibilities are compatible).

Definition 4.4. *Given a strategy expression α and an initial term $t \in T_\Sigma$, we define the Kripke structure $\mathcal{M}'_{\alpha,t} \coloneqq (\mathcal{XS}', [\twoheadrightarrow]', \{\{t \,@\, \alpha\}\}, \text{AP}_\Pi, L_\Pi \circ \text{cterm})$ where*

$$\mathcal{XS}' = \{Q \subseteq \mathcal{P}(\mathcal{XS}) \,:\, \exists\, t \in T_\Sigma \quad \forall q \in Q \;\; \text{cterm}(q) = t \;\wedge\; \exists\, q \in Q \;\; \text{valid}(q)\},$$

and for any $Q, Q' \in \mathcal{XS}'$

$$Q \; [\twoheadrightarrow]' \; Q' \quad \iff \quad \exists\, t \in T_\Sigma, a \in A \quad Q' = \{q' \,:\, q \twoheadrightarrow^a q', q \in Q, \text{cterm}(q') = t\}.$$

In summary, the states of $\mathcal{M}'_{\alpha,t}$ are sets of execution states with a common projection, and the successors of these sets are the union of the successors of their elements grouped by their subject terms and by the action.

Theorem 4.1. *$\mathcal{M}'_{\alpha,t}$ and $\mathcal{U}(\mathcal{M}, \lambda_{E(\alpha)})$ are bisimilar Kripke structures.*

Theorem 4.1 tells that $\mathcal{M}'_{\alpha,t}$ is an effective candidate to check branching-time properties on Maude specifications with strategies according to the ideas of Section 4.1. All these structures and propositions are stated in terms of labeled transition systems, while state-based logics like

LTL, CTL, and CTL* are defined on unlabeled transition systems. As we mentioned in Section 2.2, this is without loss of generality, because unlabeled transition systems can be viewed as labeled ones with a single arbitrary label. However, it is important that we forget about the labels of a labeled transition system before checking state-based properties, not only for efficiency reasons, but also for semantic ones. Otherwise, the labels will change the model semantics as the strategy continuations did at the beginning of this section. For instance, suppose a strategy `r1 ; r2 | r3 ; r4` is applied to a term t_1 with the rules $t_1 \rightarrow^{\texttt{r1}} t_2$, $t_1 \rightarrow^{\texttt{r3}} t_2$, $t_2 \rightarrow^{\texttt{r2}} t_3$, and $t_2 \rightarrow^{\texttt{r4}} t_4$. The CTL property $\mathbf{A} \bigcirc \mathbf{E} \diamond t_4$ will not be true if edge labels are considered, but it will if they are not, as it should be for state-based logics. On the contrary, for logics that operate on labeled transition systems like μ-calculus, the edge labels should be preserved and used to distinguish successor states when they are merged, because our notion of strategy $\lambda : (S \cup A)^+ \to \mathcal{P}(A \times S)$ conditions the next steps on the previous actions too. Using the same example, $\langle \texttt{r1} \rangle \langle \texttt{r4} \rangle \top$ should only be true if `r4` can be applied after `r1`. With these precautions, the following corollary claims that we can check CTL, CTL*, and μ-calculus properties, among others, using $\mathcal{M}'_{\alpha,t}$.

Corollary 4.2. $(\mathcal{M}, E(\alpha, t)) \vDash \varphi \iff \mathcal{M}'_{\alpha,t} \vDash \varphi$ *for any bisimilarity-invariant temporal property* φ.

The generated transition system $\mathcal{M}_{\alpha,t}$ is finite and its transition relation decidable if the reachable states from the initial one are finitely many (see Lemma B.14). Since merged states are the combinations of normal execution states, the number of states can grow exponentially at worst, although it would usually decrease, like in the vending machine example.

Corollary 4.3. *If the reachable states from* $t \mathbin{@} \alpha$ *by* $\to_{s,c}$ *are finitely many,* $(\mathcal{M}, E(\alpha, t)) \vDash \varphi$ *is decidable for LTL, CTL*, and μ-calculus.*

4.3 Strategy-aware extension of the Maude LTL model checker

We have extended the builtin Maude LTL model checker [EMS04] to support rewriting systems controlled by its strategy language, based on the principles of the previous sections. The original LTL model checker implements the automata-theoretic approach (explained in Section 2.3.2) clearly separating its three components: (1) the generation of a Büchi automaton for the temporal property, (2) the on-the-fly generation of an automaton for the model, and (3) the algorithm that checks whether the intersection of the previous two is empty. Strategies only restrict the executions of the model and do not interfere with the property specification, so only the second of these parts has been adapted by replacing the standard rewrite system $\mathcal{M}$ with the strategy-aware model $\mathcal{M}_{\alpha,t}$ described in Section 4.2, with the help of the infrastructure for executing strategies of the `srewrite` and `dsrewrite` commands. Consequently, a significant part of the C++ and Maude implementation of the model checker has been reused, and the interfaces of both model checkers are very similar. Users of the original model checker can use the strategy-aware one without much effort, and both can be used on the exact same module to compare the properties of the controlled and uncontrolled system. The extension is not exempt of subtleties and difficulties that are described in more detail in Section 4.7.

Figure 4.4 outlines how strategy-aware models are typically prepared for model checking, where some modules available in the Maude prelude or provided in the `model-checker.maude` file of the model checker distribution are involved. The process is done in much the same way as explained in the Maude manual [CDE+20] for the original model checker. The input model is given by a system module `M` describing the uncontrolled system and a strategy module `SM` defining one or more strategies to control its behavior.[3] In order to specify atomic propositions,

[3]The separation of the modules `M` and `SM` in the model specification is a matter of style. In general, we propose specifying the static model representation and the rules in a system module `M`, and describing how they are controlled in a strategy module `SM` being a protecting extension of `M`.

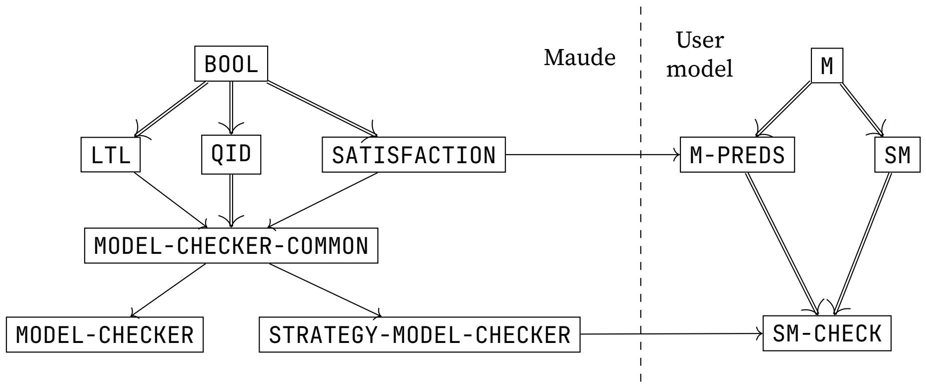


Figure 4.4: Structure of the strategy model checker modules.

a module `M-PREDS` is defined as a protecting extension of `M` where the builtin `SATISFACTION` module is included, providing the formal sorts `State` for the model states and `Prop` for atomic propositions, and the symbol $\_ \models \_$ to define with equations whether these atomic propositions are satisfied in each state.

```
fmod SATISFACTION is
  protecting BOOL .
  sorts State Prop .
  op _⊨_ : State Prop → Bool [frozen] .
endfm
```

The intended sort of states in `M` is defined as a subsort of `State` in `M-PREDS`, along with the declaration and definition of the atomic propositions, i.e. the signature Π and the equations D mentioned in Section 4.2. Finally, both `M-PREDS` and `SM` are gathered in a strategy module `SM-CHECK` that includes the module `STRATEGY-MODEL-CHECKER`. This module is the traditional entry point to the model checker via a special `modelCheck` operator that receives the problem data and reduces to its verification result:

```
op modelCheck : State Formula Qid QidList Bool
                ⇝ ModelCheckerResult [special(...)] .
```

The first and second arguments coincide with the `modelCheck` operator of the classical model checker: the initial term and the LTL formula to be checked, with the syntax specified in the `LTL` module of the original model checker and using the atomic propositions in `M-PREDS`.

```
fmod LTL is
  sort Formula .
  ops True False : → Formula [ctor ...] .
  op ~_ : Formula → Formula [ctor prec 53 ...] .
  op _/\_ : Formula Formula → Formula [comm ctor prec 55 ...] .
  op _\/_ : Formula Formula → Formula [comm ctor prec 59 ...] .
  op O_ : Formula → Formula [ctor prec 53 ...] .
  op _U_ : Formula Formula → Formula [ctor prec 63 ...] .
  op _->_ : Formula Formula → Formula [prec 65 ...] .
  op <>_ : Formula → Formula [prec 53 ...] .
  op []_ : Formula → Formula [prec 53 ...] .
  *** [...]
endfm
```

The strategy that must control the system is selected in its third argument by providing its name as a quoted identifier. This name must refer to a strategy without arguments available in the `SM` module.[4] Requiring a name is not a serious limitation, because any strategy expression can be given one in a strategy module, although it may sometimes be uncomfortable. Alternatively, an arbitrary strategy expression can be provided when using the unified Maude model-checking tool `umaudemc`, explained in Section 4.4, which has some other advantages. The fourth and fifth arguments of the `modelCheck` operator enable some optional features that are explained in Section 4.3.1. Overloads are defined so that these arguments can be omitted.

Reducing the term `modelCheck(`t`,` φ`, '`$sname$`)` triggers the evaluation of $(\mathcal{M}, E(sname, t)) \vDash \varphi$ or equivalently $\mathcal{M}_{sname,t} \vDash \varphi$. If the property is satisfied, the operator is reduced to the constant `true` of sort `Bool`. In case the property does not hold, the term `counterexample(`π`,` ξ`)` is returned, consisting of a path π and a cycle ξ such that $\pi\xi^\omega \in E(sname, t)$ is an execution allowed by the strategy that refutes the formula. Like in the standard model checker, these sequences are a juxtaposition of transitions `{`t`,` r`}` where t is a term and r describes the transition that rewrites this term into the next one. This tag is the name of the rule that has been applied or the constant `unlabeled` if it does not have one. Moreover, in the last transition of the cycle, r can take the value `solution` to indicate that a finite strategy execution refutes the given property.

Let us illustrate all this with the simple and classical example of the river-crossing puzzle, where a shepherd needs to cross a river carrying a wolf, a goat, and a cabbage. The only means is using a boat that only the shepherd can drive and with room for only one more passenger. Shipping the companions of the shepherd one by one would be a solution, but the wolf would eat the goat and the goat would eat the cabbage as soon as the shepherd leaves them alone. The static part of the puzzle is specified in the following functional module.

```
fmod RIVER-DATA is
  sorts Being Side Group River .
  subsorts Being Side < Group .

  ops shepherd wolf goat cabbage : -> Being [ctor] .
  ops left right : -> Side [ctor] .
  op __  : Group Group -> Group [ctor assoc comm prec 40] .
  op _|_ : Group Group -> River [ctor comm] .

  vars G1 G2 : Group .

  op initial : -> River .
  eq initial = left shepherd wolf goat cabbage | right .

  op risky : River -> Bool .
  eq risky(shepherd G1 | G2 wolf goat ) = true .
  eq risky(shepherd G1 | G2 goat cabbage) = true .
  eq risky(G1 | G2) = false [owise] .
endfm
```

Characters are declared as constants of sort `Being` and put together in a `Group` set at both sides of the `River`. The operator `|` used to separate these sides is commutative because this will simplify the definition of the rules, but we should indicate which the initial and the destination side were so as to know if the puzzle is solved. This is done by the `left` and `right` constants of sort `Side`. In the `initial` position of the puzzle all characters are on the left border. Finally, the predicate `risky` identifies the states in which some being is at risk of being eaten.[5]

The possible moves of the game are specified using rules in the following `RIVER` module. There is a rule to cross the river for each character, and there are two more rules `wolf-eats`

[4]The `modelCheck` argument cannot receive an arbitrary strategy expression because the object-level strategy language is not represented in Maude itself.

[5]As explained in Section 2.5, equations annotated with the `owise` attribute are executed only after all equations without this attribute have failed.

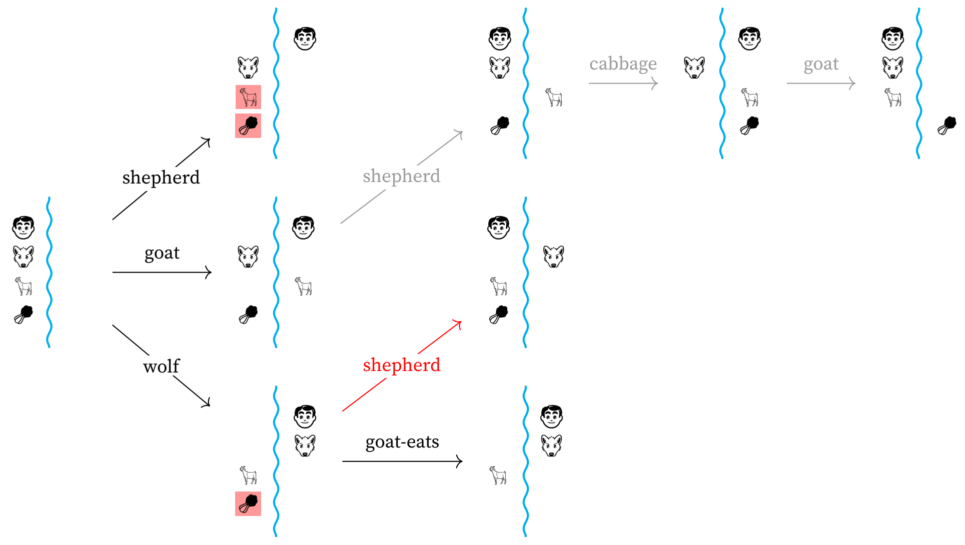


Figure 4.5: Partial rewriting tree for the river-crossing puzzle.

and `goat-eats` that make the mentioned animal eat its colleague one trophic level below.

```
mod RIVER is
  protecting RIVER-DATA .

  vars L R : Group .

  rl [alone]   :         shepherd L | R ⇒ L | R shepherd .
  rl [wolf]    :    shepherd wolf L | R ⇒ L | R shepherd wolf .
  rl [goat]    :    shepherd goat L | R ⇒ L | R shepherd goat .
  rl [cabbage] : shepherd cabbage L | R ⇒ L | R shepherd cabbage .

  rl [wolf-eats] :    wolf goat L | R shepherd ⇒ wolf L | R shepherd .
  rl [goat-eats] : goat cabbage L | R shepherd ⇒ goat L | R shepherd .
endm
```

This rewriting system does not guarantee that the rules of the game are respected, since escaping from a risky state without applying `wolf-eats` or `goat-eats` is possible, as shown in the lower branch of Figure 4.5. This suggests that the eating rules must be applied eagerly before any movement rule is executed again. In a previous version of this example in Maude [PMV05], the rules `wolf-eats` and `goat-eats` were equations. Since rewrite rules are applied in Maude after reducing the terms to their canonical forms using equations, eating would take place eagerly before moving as desired. However, functional specifications are assumed to be confluent, terminating, and coherent with the rules in Maude, and this implies that the equational reduction of a term should not prevent the application of a rule. This property would not be satisfied if eating rules were equations, since the application of the equation `wolf-eats` would hinder the application of the rule `goat` in the canonical form, even though it would be applicable in the unreduced term. Fortunately, strategies can be used to specify the required precedence, and it is done in the following strategy module.

```
smod RIVER-STRAT is
  protecting RIVER .

  strats oneCrossing eating eatb4cross cross&eat solved @ River .
  sd oneCrossing := alone | wolf | goat | cabbage .
  sd eating      := wolf-eats | goat-eats .
  sd eatb4cross  := eating or-else oneCrossing .
  sd cross&eat   := oneCrossing ; eating ! .
  sd solved      := match left | right shepherd wolf cabbage goat .

  strats eagerEating eagerEating2 safe @ River .

  sd eagerEating := solved ? idle : (eatb4cross ; eagerEating) .
  sd eagerEating2 := solved ? idle : (cross&eat ; eagerEating2) .
  sd safe := solved ? idle : (oneCrossing ; not(eating) ; safe) .
endsm
```

The strategies eagerEating and eagerEating2 are two alternative ways of ensuring that eating happens before moving. They are defined using some simple auxiliary strategies like oneCrossing that applies any of the crossing rules once, eating that does the same for the eating rules, solved that detects with a test whether the solution has been found, eatb4cross that executes oneCrossing only if eating has failed, and finally cross&eat that moves once and then eats exhaustively. The recursive definitions of eagerEating and eagerEating2 keep respectively applying eatb4cross and cross&eat indefinitely or until the solution of the puzzle is found, hence respecting the rules of the puzzle. The strategy safe is more restrictive and describes all paths that completely avoid risky states, because the strategy execution is aborted by not(eating) when they are visited.

According to the schema of Figure 4.4, the modules RIVER and RIVER-CROSSING play the role of the M and SM modules, as they are the model where properties will be checked. The next step is declaring and defining the atomic propositions on which temporal properties will be based in the protected extension RIVER-PREDS of RIVER.

```
mod RIVER-PREDS is
  protecting RIVER .
  including SATISFACTION .

  subsort River < State .
  ops goal risky death : -> Prop [ctor] .

  var  R    : River .
  vars G G' : Group .

  eq left | right shepherd wolf goat cabbage |= goal = true .
  eq R |= goal = false [owise] .
  eq G cabbage | G' goat |= death = false .  *** Only cabbage and goat may die
  eq G cabbage goat | G' |= death = false .
  eq R |= death = true [owise] .
  eq R |= risky = risky(R) .
endm
```

The sort River is selected as the sort of the model states by the **subsort** declaration, and three atomic propositions are defined: goal that holds only in the goal configuration of the puzzle, death that is satisfied when any of the characters has disappeared, and risky that labels the risky states.

Finally, a module like SM-CHECK should combine the rewriting and strategic specification with the STRATEGY-MODEL-CHECKER module that gives access to the model checker.

```
smod RIVER-CHECK is
  protecting RIVER-PREDS .
```

```
   protecting RIVER-STRAT .
   including STRATEGY-MODEL-CHECKER .
   including MODEL-CHECKER .
endsm
```

The module MODEL-CHECKER of the original model checker is also included, because they are compatible and we will check some properties on the uncontrolled system too. Notice that disgregating the specification in this panoply of modules is a matter of style, and all could have been written in a single module.

One of the most essential properties of the river-crossing puzzle is that risky states are always resolved to death states, in which a character has been actually eaten. The LTL formula $\Box(\mathit{risky} \to \Diamond\, \mathit{death})$ expresses a weaker requirement, which is not even satisfied in the uncontrolled system.

```
Maude> red modelCheck(initial, [] (risky -> <> death)) .
rewrites: 38
result ModelCheckResult: counterexample(
   {left shepherd wolf goat cabbage | right,'alone}
    ...
   {left shepherd cabbage | right wolf goat,'cabbage},
   {left | right shepherd wolf goat cabbage,'alone}
   {left shepherd | right wolf goat cabbage,'alone})
```

However, when the system is controlled by the eagerEating or eagerEating2 strategies, the stronger property in which the risky state is resolved immediately $\Box(\mathit{risky} \to \bigcirc\, \mathit{death})$ is satisfied:

```
Maude> red modelCheck(initial, [] (risky -> O death), 'eagerEating) .
rewrites: 158
result Bool: true
Maude> red modelCheck(initial, [] (risky -> O death), 'eagerEating2) .
rewrites: 242
result Bool: true
```

In general, more restrictive strategies satisfy more linear-time properties, so the formulae satisfied by eagerEating must be satisfied by safe. Moreover, under this last strategy the model satisfies the property $\Box \neg\, \mathit{risky}$, which ensures that no risky state is ever reached:

```
Maude> red modelCheck(initial, [] ~ risky, 'safe) .
rewrites: 71
result Bool: true
```

The specification of safe makes the model checker visit some risky states because oneCrossing is executed unconditionally. However, these paths are immediately discarded by the strategy **not**(eating) that follows oneCrossing in its definition, and so, they do not count as states of the model.

Moreover, we can use the model checker to find a solution to the puzzle. If we check $\Box \neg\, \mathit{goal}$, a counterexample will be a derivation in which *goal* is reached. This path could not necessarily be the shortest solution, but in this case it is:

```
Maude> set verbose on .
Maude> red modelCheck(initial, [] ~ goal, 'eagerEating) .
ModelChecker: Property automaton has 2 states.
StrategyModelCheckerSymbol: Examined 31 system states (31 real).
rewrites: 95
result ModelCheckResult: counterexample(
  {right | left shepherd wolf goat cabbage,'goat}
  {left wolf cabbage | right shepherd goat,'alone}
  {right goat | left shepherd wolf cabbage,'wolf}
  {left cabbage | right shepherd wolf goat,'goat}
  {right wolf | left shepherd goat cabbage,'cabbage}
  {left goat | right shepherd wolf cabbage,'alone}
  {left shepherd goat | right wolf cabbage,'goat},
  {left | right shepherd wolf goat cabbage,solution})
```

If the original model checker were used with the same property, a counterexample of length 12 that reaches the result in 5 steps is obtained. However, it visits the risky state `left goat cabbage | right shepherd wolf` in its first step, so it is not a solution for the puzzle. The number of states of the property and system automata are shown by the interpreter if the verbose mode of Maude is enabled with `set verbose on`.

4.3.1 Advanced features

The following paragraphs are dedicated to the fourth and fifth arguments of the `modelCheck` operator, and the equivalent options of the `umaudemc` tool, which deviate intentionally from the model specified by the semantics in Section 3.5.

Opaque strategies. The main principle of our understanding of model checking for systems controlled by strategies is that their executions are a subset of those of the original system. In addition to its theoretical convenience, the principle has practical implications when model checking linear-time properties, since refuting a property on a system controlled by a strategy refutes the property for the original system. However, it can be sometimes useful to deviate from this principle and consider the execution of some strategies as atomic steps, rather than the rule rewrites they consist of. Such a sequence of $\twoheadrightarrow$ steps can be seen as a single step, with transitions linking the state where the strategy has been called to those in which its execution concludes.

Strategies whose executions are considered atomic are called *opaque strategies* and passed to the model checker as a list of strategy names in its fourth argument. The list cannot discriminate between homonym strategies with a different signature, but renaming the desired strategy using the Maude renaming support is easier than admitting signature specifications there. In the `modelCheck` result, opaque strategies appear as `opaque(`s`)` in place of the rule name where s is the name of the strategy.

Strategies are suitable for representing parallel rewriting by gathering multiple rule rewrites in a single strategy application. Parallel steps and other meaningful aggregations of rewrites can be seen as the atomic transitions of the model by means of opaque strategies. An example is in Chapter 8, where membrane systems are represented in rewriting logic using the strategy language. The atomic step of this unconventional computational model consists of multiple phases of parallel rule applications, and it is specified as an opaque strategy. This effectively hides the serialization of parallel rewrites and the meaningless intermediate states between phases, while making the *next* operator semantically accurate. Moreover, opaque strategies can be used to test properties at different levels of granularity, as shown in Sections 6.4 and 7.3.

In the river-crossing example, the `eagerEating` strategy guarantees that risky states are immediately resolved by eating rules. However, the puzzle controlled by `eagerEating` does not satisfy the property $\Box\neg\,risky$, since these states are actually visited.

```
Maude> red modelCheck(initial, [] ~ risky, 'eagerEating2) .
rewrites: 15
result ModelCheckResult:  counterexample(
  {right | left shepherd wolf goat cabbage,'alone}
  {right shepherd | left wolf goat cabbage,'wolf-eats}
  {right shepherd | left wolf cabbage,'alone},
  {right | left shepherd wolf cabbage,'alone}
  {right shepherd | left wolf cabbage,'alone})
```

Instead, the `cross&eat` strategy called by `eagerEating2` can be seen as an atomic step using opaque strategies, making the intermediate risky states invisible for the model checker. The property is then satisfied:

```
Maude> red modelCheck(initial, [] ~ risky, 'eagerEating2, 'cross&eat) .
rewrites: 136
result Bool: true
```

Even if this feature were not explicitly supported by the model checker, opaque strategies could be implemented using a rule with a rewriting condition

```
rl [opaque] : X => Y if X => Y [nonexec] .
```

where `X` and `Y` are variables of the `State` kind. When applying the strategy expression α as `opaque{`α`}`, its execution will be seen as an atomic step, as an application of the `opaque` rule. Hence, declaring a strategy *sl* as opaque in the `modelCheck` term would produce essentially the same effect as wrapping every strategy call to *sl* as indicated before. This alternative is more general and selective, but it requires extending the base rewrite system with this additional rule, and modifying the strategy expressions to change the opacity of a strategy.

Biased matchrew as a form of partial order reduction. The intended meaning of the `matchrew` family of combinators is the parallel rewriting of the matched subterms using the specified strategies. However, executions are described as linear sequences of rule rewrites, so those coming from the different subterms must be ordered. The small-step semantics permits the progress of any subterm at any moment, hence considering all possible interleavings of the subterm rewriting paths as executions of the `matchrew`. This is semantically accurate but computationally expensive, since even in the case of only two subterms with a single rewriting path allowed for each of them, this yields the binomial coefficient $n + m$ over n of interleaved executions where n and m are the length of these paths. When the model checker users know that the ordering of the subterm paths does not affect the satisfaction of the property in question, they can choose to exhibit only one representative to the model checker as a form of partial order reduction, avoiding the generation of the full set of combinations. The $n + m$ over n executions of the small hypothetical case above are reduced to a single one. Whether this *biased matchrew* feature is used or not is specified in the optional Boolean argument of the `modelCheck` operator. By default, its value is `true`, so it is enabled. Specifically, biased executions have their rewrites ordered like the subterms in the `matchrew` combinator, from left to right, so that all the rewrites in the k-th subterm occur before those in the $(k + 1)$-th.

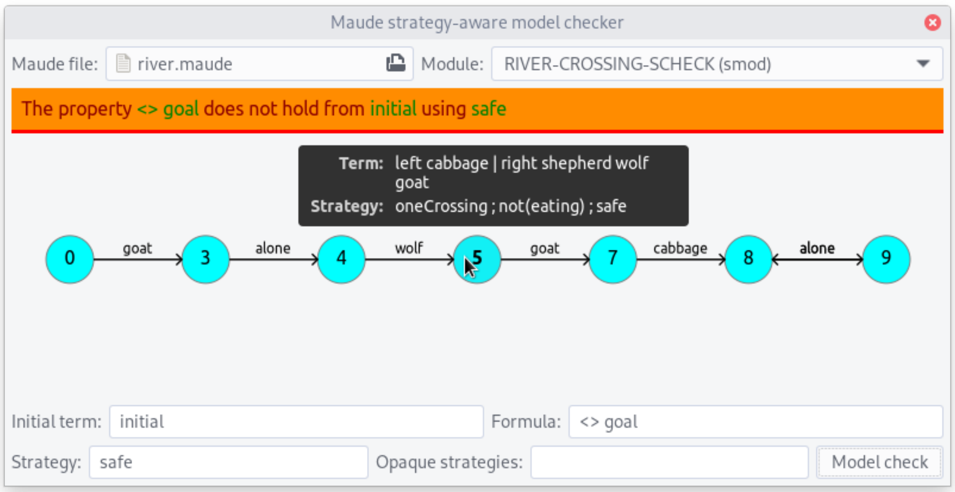
For example, we can informally consider a system with two processes and a shared resource, and the following `matchrew` as part of the strategy that controls the system:

```
matchrew < P1, P2, SR > by P1 using step ! , P2 using step ! .
```

Supposing that this strategy advances the processes until they need the shared resource, and the property refers only to the shared resource ownership, the property will be satisfied or refuted regardless of the interleaving of the processes states. However, if the property refers to certain relationships between the two processes, ignoring some executions may miss counterexamples that refute the property.

4.4 Model checking using external tools

The just described extension of the Maude LTL model checker for strategy-controlled specifications generates as part of its job a labeled transition system, the $\mathcal{M}_{\alpha,t}$ of Section 4.2. With the


Maude strategy-aware model checker

Maude file: river.maude Module: RIVER-CROSSING-SCHECK (smod)

The property <> goal does not hold from initial using safe

Term: left cabbage | right shepherd wolf goat
Strategy: oneCrossing ; not(eating) ; safe

Initial term: initial Formula: <> goal

Strategy: safe Opaque strategies:

Figure 4.6: Counterexample shown in the graphical interface of `umaudemc`.

adaptations described in Section 4.2.1, this LTS can be transformed into $\mathcal{M}'_{\alpha,t}$, where branching-time properties can be properly checked. Thanks to the modular design of the original model checker, adopted by our extension, this model is exposed as an abstract Kripke structure where the successors and the atomic properties satisfied by a state can be queried using C++ functions. Hence, model checking properties in other logics only requires implementing their algorithms and the adaptations on top of this interface. However, instead of writing our own model-checking algorithms, we have found convenient to reuse already used and tested implementations for the target logics, since they are ultimately based on Kripke structures. A good candidate is the language-independent model checker LTSmin [KLM+15], which is able to efficiently interact with our Kripke-like representation of the model on the fly at the C++ level and supports very general logics like CTL* and μ-calculus. In addition, we have established connections with other model checkers like NuSMV [CCG+02], the `pyModelChecking` [Cas20] library, and Spot [DLF+16], and we have also written our own implementation of a μ-calculus algorithm. Additional logics and backends can be added without much effort using this approach. More details about these connections are given at the end of this section.

Aiming at discharging users from learning the particular syntax and mode of operation of the different backends, a common and simplified interface is provided by the *unified Maude model-checking tool* `umaudemc` [Rub21b].

4.4.1 The unified model-checking tool `umaudemc`

This program can be used from the command line, from a graphical user interface shown in Figure 4.6, or even programmatically as a library. In either case, the tool receives the model-checking problem data in a format that does not depend on any particular model checker and outputs the result in equal terms. The syntax of its command-line invocation is the following:

`umaudemc check` ⟨ *file name* ⟩ ⟨ *initial term* ⟩ ⟨ *formula* ⟩ [⟨ *strategy* ⟩]

Square brackets surround the strategy argument to indicate that it can be omitted, since properties can also be checked on uncontrolled Maude specifications with all the supported backends. The formula is expressed in a syntax that extends the predefined Maude `LTL` module with operators for CTL* and μ-calculus. For the first logic, the only new constructors are the universal `A_` and existential `E_` path quantifiers. For the μ-calculus, the syntax is extended with the universal modalities `[_]_` and `[.]_`, the existential modalities `<_>_` and `<.>_`, the fixed-point operators `mu_._` and `nu_._`, and variables.

```
*** CTL and CTL*
op A_ : Formula -> Formula [ctor prec 53] .
op E_ : Formula -> Formula [ctor prec 53] .

*** mu-calculus
subsort @MCVariable@ < Formula .

op <.>_  : Formula -> Formula [ctor prec 53 format (c o d)] .
op [.]_  : Formula -> Formula [ctor prec 53 format (c d d os d)] .
op <_>_  : @ActionSpec@ Formula -> Formula [ctor prec 53 ...] .
op [_]_  : @ActionSpec@ Formula -> Formula [ctor prec 53 ...] .
op mu_._ : @MCVariable@ Formula -> Formula [ctor prec 64] .
op nu_._ : @MCVariable@ Formula -> Formula [ctor prec 64] .

*** Action lists
sorts @Action@ @ActionSpec@ @ActionList@ .
subsorts @Action@ < @ActionList@ < @ActionSpec@ .
op __ : @ActionList@ @ActionList@ -> @ActionList@ [ctor assoc] .
op ~_ : @ActionList@ -> @ActionSpec@ [ctor] .
op opaque : @Action@ -> @ActionList@ [ctor] .
```

Modalities are generalized so that they can take one or more rule or opaque strategy labels of the module as actions, separated by space. This follows the widespread notation $[C]\,\varphi \coloneqq \bigwedge_{a\in C}[a]\,\varphi$ and $\langle C\rangle\,\varphi \coloneqq \bigvee_{a\in C}\langle a\rangle\,\varphi$. In case C is the complete set of actions, a dot can be written instead. The complement of the list of actions can be specified by preceding it with the negation symbol `~`. Variables for μ-calculus can be any token that does not interfere with the other elements in the formula. The sorts `@Action@` and `@MCVariable@` are populated at the metalevel before parsing, based on the rule labels and strategy names of the target module, and on a previous scan of the formula. The `opaque` operator can be used to surround opaque strategy labels, but it is only required when there is a homonym rule. The `umaudemc` tool parses the input formula within this Maude signature, deduces the least-general logic this formula belongs to, and then calls the appropriate backend with the appropriate configuration.

To illustrate its usage, we will check some branching-time properties of the river-crossing puzzle. The CTL formula $\mathbf{A}\,\square\,\mathbf{E}\,\lozenge\,\mathit{goal}$ expresses that every state of the river-crossing puzzle can be continued to a solution. This formula is satisfied when the system is controlled by the `safe` strategy, but not when using the `eagerEating` strategy or when the system runs uncontrolled.

```
$ umaudemc check river.maude initial 'A [] E <> goal' safe
The property is satisfied in the initial state
(16 system states, 264 rewrites).

$ umaudemc check river.maude initial 'A [] E <> goal' eagerEating
The property is not satisfied in the initial state
(35 system states, 3292 rewrites).

$ umaudemc check river.maude initial 'A [] E <> goal'
The property is not satisfied in the initial state
(36 system states, 3058 rewrites).
```

The reason is that no solution can be reached once a character has been eaten, which may happen in the last two cases. Counterexamples are only shown if the selected backend supports them, and the `-c` flag can be used to prefer one of these. The following counterexample confirms our explanation for the refutation of the last property.[6] Transitions are labeled by default with the full statement that caused them, although the counterexamples format is highly configurable.

```
$ umaudemc check river.maude initial 'A [] E <> goal' -c
The property is not satisfied in the initial state
(36 system states, 125 rewrites)
| right | left shepherd wolf goat cabbage
v rl G' | shepherd G => G | shepherd G' [label alone] .
| right shepherd | left wolf goat cabbage
v rl shepherd G' | wolf goat G => shepherd G' | wolf G [label wolf-eats] .
O right shepherd | left wolf cabbage
```

However, the property $\mathbf{A}\,\square\,(risky \lor death \lor \mathbf{E}\,\Diamond\,goal)$ holds under the eagerEating strategy.

```
$ umaudemc check river.maude initial \
    'A [] (risky \/ death \/ E <> goal)' eagerEating
The property is satisfied in the initial state
(35 system states, 982 rewrites).
```

We can also check μ-calculus properties, like the fact that the only initial movement not leading to a risky state is goat:

```
$ umaudemc check river.maude initial \
    '[ alone wolf cabbage ] risky /\ < goat > ~ risky'
The property is satisfied in the initial state
(5 system states, 13 rewrites, 7 game states).
```

We wonder if the goal can be reached without moving the goat again: this is the μ-calculus formula $[\texttt{goat}]\,(\mu Z.\, goal \lor \langle \texttt{alone wolf cabbage} \rangle\, Z)$, whose fixed-point subformula describes the states where the goal can be reached using any sequence of moves other than goat. The answer is negative if the rules of the game are respected, like in the eagerEating strategy:

```
$ umaudemc check river.maude initial \
    '[ goat ] (mu Z . goal \/ < ~ goat > Z)' eagerEating
The property is not satisfied in the initial state
(35 system states, 110 rewrites, 157 game states).
```

We have replaced the list of labels alone wolf cabbage by ~ goat to illustrate the complement notation for actions. These are not exactly the same, because the complement of goat also includes the rules wolf-eats and goat-eats, but they do not change the satisfaction of the property. On the contrary, the uncontrolled system satisfies the formula, since it can pass by forbidden states:

```
$ umaudemc check river.maude initial \
    '[ goat ] (mu Z . goal \/ < ~ goat > Z)'
The property is satisfied in the initial state
(33 system states, 84 rewrites, 97 game states).
```

In order to properly check branching-time properties, the umaudemc tool automatically enables the adaptations discussed in Section 4.2.1 according to the input formula. However, the --purge-fails and --merge-states arguments can overwrite the defaults. For efficiency, it may be interesting to disable them when we know in advance that they will not have any effect on a particular model. We will do it here just to illustrate their implications. Coming back to the vending machine example of Section 4.2.1, we claimed that the CTL property $\mathbf{A}\bigcirc\mathbf{E}\,\Diamond\, hasCake$ is

[6]For a branching-time logic, counterexamples can be provided for purely universal formulae and examples for purely existential formulae. In case both quantifications are mixed, a prefix of the path until the second quantifier applies can be given. This is marked in umaudemc by an O in the last line of the counterexample.

not satisfied on the original graph when the system is controlled by the strategy α, but it is when controlled by the equivalent strategy β. This can be checked by disabling the `merge-states` adaptation:

```
$ umaudemc check vending.maude initial 'A O E <> hasCake' \
  'put1 ; apple | put1 ; put1 ; cake' --merge-states=no
The property is not satisfied in the initial state
(6 system states, 72 rewrites)

$ umaudemc check vending.maude initial 'A O E <> hasCake' \
  'put1 ; (apple | put1 ; cake)' --merge-states=no
The property is satisfied in the initial state
(5 system states, 60 rewrites).
```

However, when states are properly merged, the property is satisfied for both strategy expressions as follows from Corollary 4.2:

```
$ umaudemc check vending.maude initial 'A O E <> hasCake' \
  'put1 ; apple | put1 ; put1 ; cake'
The property is satisfied in the initial state
(6 system states, 52 rewrites).
```

The transition systems generated for each model with the different adaptations can be observed with the `umaudemc graph` command. Something similar to Figure 4.3 would be obtained for these examples.

Linear-time formulae can also be checked using the `umaudemc` tool. They are checked by default using the builtin Maude LTL model checker, which is usually the most convenient, but the majority of alternative tools support this logic too. Shorter counterexamples can be sometimes obtained with the Spot and NuSMV backends:

```
$ umaudemc check river.maude initial '<> goal' safe --backend spot
The property is not satisfied in the initial state
(16 system states, 49 rewrites, 1 Büchi state)
| right | shepherd wolf goat cabbage left
v rl R | shepherd goat L => L | R shepherd goat [label goat] .
| | shepherd goat right | wolf cabbage left
| v rl R | shepherd L => L | shepherd R [label alone] .
| | goat right | shepherd wolf cabbage left
| v rl R | shepherd L => L | shepherd R [label alone] .
< v
```

Simplifying the counterexample output and obtaining information that is not directly available through the traditional interface is possible by means of different options. For example, atomic propositions can be checked on the states, and the next pending strategy can be printed.

```
$ umaudemc check river.maude initial '<> goal' safe --backend spot
  --elabel '%l' --slabel 'goal={%t |= goal} @ %s'
The property is not satisfied in the initial state
(16 system states, 49 rewrites, 1 Büchi state)
| goal=false @ oneCrossing ; not(eating) ; safe
v goat
| | goal=false @ oneCrossing ; not(eating) ; safe
| v alone
| | goal=false @ oneCrossing ; not(eating) ; safe
| v alone
< v
```

More details about these features are shown with the `--help` flag.

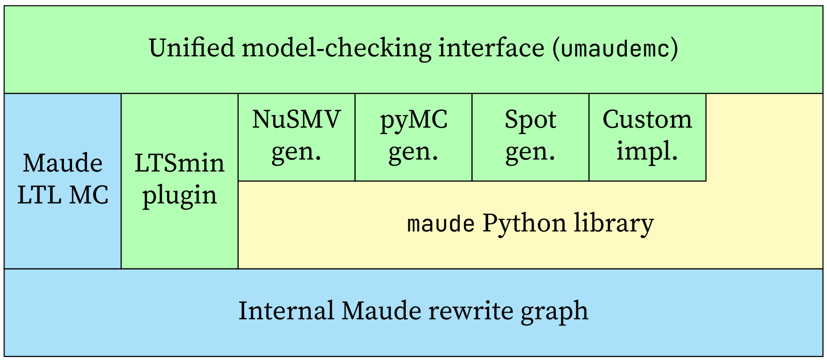


Figure 4.7: Architecture of the `umaudemc` model-checking tool.

4.4.2 The architecture of `umaudemc`

As we have seen with the examples of the previous section, the `umaudemc` tool allows checking temporal properties on both standard and strategy-controlled Maude specifications regardless of which model-checking backend is doing the job behind the scenes. All of them rely on the internal Maude rewrite graphs used by the Maude LTL model checker [EMS04] and by our extension for strategy-controlled systems.[7] This is outlined in Figure 4.7, where some backends access these graphs directly while others use a Python library called `maude`, which is described in Appendix A. This library exposes all relevant Maude entities and operations as objects and methods in Python by directly interacting with the Maude implementation at the binary level, including the strategy-controlled and the standard rewrite graphs. Models are generated for the various supported backends by exploring these graphs and evaluating atomic propositions on them.

The `maude` library is also used directly by the `umaudemc` tool to process the problem data, the verification results and the counterexamples, and to produce printable graphs of the models. The extended language of temporal properties admitted by the tool is specified in a Maude module, parsed using the library, and translated to the syntax of the temporal properties supported by the selected backend. Whether the adaptations of Section 4.2.1 are applied or not is also decided depending on the problem data, and they are implemented in C++ inside the LTSmin plugin or in Python for the backends based on the `maude` library. The `umaudemc` tool will detect which backends are installed and call the most convenient for each supported logic, although the search order can be changed with the `--backend` option. In addition to the Maude LTL model checker and LTSmin, which is described in Section 4.4.3, the available backends are:

- NuSMV [CCG+02], which supports LTL and CTL. The model is prepared as a low-level specification file in the NuSMV format, the model checker is called on this file, and the results are parsed from its output. It calculates counterexamples for CTL properties too.

- `pyModelChecking` [Cas20] is a Python library with textbook model-checking implementations for LTL (by the tableau method), CTL, and CTL*. The Kripke structure is constructed as a Python object from the Maude model.

- Spot [DLF+16] is a C++ framework for LTL and ω-automata manipulation with a Python library. Kripke structures are built for checking LTL properties as in the other backends, but more complex ω-automata are supported. These are used in Section 4.5 to handle the Kleene star semantics of the iteration. It also admits on-the-fly model checking, but not through the Python interface.

[7]In the implementation, the uncontrolled and strategy-controlled Maude rewrite graphs correspond to the C++ classes `StateTransitionGraph` and `StrategyTransitionGraph` respectively.

	LTL	CTL	CTL*	μ-calculus	Lines
Extended Maude	on-the-fly				1200
LTSmin	on-the-fly	√	√	√	1140
pyModelChecking	tableau	√	√		147
NuSMV	tableau	√			199
Spot	automata				203
Builtin		√		√	400

Table 4.1: Logics supported by the backends in `umaudemc`.

- Our own Python implementation of the μ-calculus model-checking procedure in [BW18], using the Zielonka algorithm [Zie98] for parity game solving. CTL properties are also supported by a direct translation of the formulae.

Table 4.1 summarizes which logics can be checked with each backend. Although LTSmin supports all logics we have considered, other model checkers are easier to install, provide more informative output, or exhibit better performance in some cases despite their less efficient connection. Performance is discussed in Chapter 10. Adding connections to other tools and logics is relatively simple, as suggested by the number of code lines written for each backend in Table 4.1, since the models described in Section 4.2 are easily accessible and compatible in principle with any logic.

The `umaudemc` tool can also be extended from the user point of view. It provides a small API to check properties and generate graphs from other Python programs. This is useful for making branching-time model checking available to metalanguage applications like those in Chapter 8 and Sections 7.2 and 7.4.

4.4.3 The LTSmin language plugin

LTSmin [KLM$^+$15] is a collection of generic model-checking programs that can operate on models expressed in different specification languages. These models are exposed as Kripke structures by some builtin or pluggable language modules using its *Partitioned Next State Interface* (PINS). In order to check properties with this toolset, we have implemented a language module for Maude. The module `libmaudemc` is a shared C library linked with the implementation of Maude[8] that exports the functions required by the PINS interface. Model checking a temporal property using LTSmin and the Maude plugin consists of the following steps:

1. The `pins2lts-*` model-checking tools of LTSmin are called with the problem data and with a `--loader` argument indicating the path of the Maude plugin. The Maude language module is loaded in memory using the POSIX's `dlopen` API,[9] so that its exported functions and global variables required by the PINS interface can be accessed. One of these functions is called to pass the Maude-specific problem data to the plugin (the initial term, the strategy, and some other parameters) and prepare the Kripke structure that will be made available to the model-checking algorithms.

2. When the model-checking algorithm for the given logic wants to know if an atomic property p is satisfied in a state, it calls the `state_label` function of the plugin that evaluates the term $\mathrm{cterm}(Q) \models p$ and returns its Boolean result. When the model checker requires the successors of a state, it calls the `next_state` function that enumerates them with their corresponding edge labels.

[8] Maude is usually distributed as a single binary, but we have built it as a shared library `libmaude` to distribute the interpreter and this plugin together without including twice the same executable code.

[9] The `dlopen` API allows loading shared libraries at run time with the `dlopen` function, and obtaining pointers to their symbols –functions and variables– with `dladdr`.

3. The verification result is printed to the terminal. For the μ-calculus, a parity game is generated instead, which has to be solved by an external tool from the mCRL2 project [BGK⁺19].

The Kripke structures presented to LTSmin are $\mathcal{M}$, $\mathcal{M}_{\alpha,t}$ or $\mathcal{M}'_{\alpha,t}$ depending on whether the model is controlled by a strategy or not, and on the `--purge-fails` and `--merge-states` arguments passed to the language plugin. In fact, the `next_state` function called in the second step is chosen at the beginning from a small set of alternative functions that implement the adaptations described in Section 4.2.1 for state-based or edge-based branching-time logics. This choice could be inferred from the temporal formulae to be checked, but this information is never passed to the language plugin. Atomic propositions are written as quoted and escaped terms in the input formula, but they must also be supplied with the `--aprops` argument to be registered in advance by the language plugin. The set of potential atomic propositions may be infinite, as they are regular Maude operators with parameters, so this requirement cannot be avoided without the possibility of reading the formula. These disadvantages and the choice of an appropriate LTSmin command for a given property are avoided when using the `umaudemc` utility. Among the different programs included in LTSmin, `umaudemc` invokes its sequential explicit-state model checker `pins2lts-seq` for LTL and μ-calculus properties with action specifications,[10] and the symbolic model checker `pins2lts-sym` for CTL, CTL*, and μ-calculus properties without action specifications (i.e. using only the `<.>` and `[.]` modalities).

The Maude language module does not take full advantage of LTSmin. Its PINS interface allows representing states as vectors of integer indices and declaring dependencies between their entries, so that the model-checking algorithms can use them for better efficiency and parallelization. However, a state in our plugin is a single index to the internal Maude rewrite graph. Automatically partitioning an arbitrary Maude specification and inferring dependencies between the resulting parts seems to be a very complex task.

4.5 The iteration strategy and the Kleene star

As we have explained in Section 3.6.1, the iteration of the Maude strategy language is not seen as a Kleene star by the operational semantics of Section 3.5 and by the model checkers on the previous sections. Instead, $\alpha\star$ behaves like the ω-regular expression $\alpha^* \mid \alpha^\omega$, allowing the body to be executed indefinitely. This inaccuracy is not usually a problem, since the star combinator is commonly used when only a finite number of iteration rounds is possible; and when this number is not bounded, an impractical infinite search space will usually be produced. Only when cyclic executions are possible within an iteration, making this combinator behave as the Kleene star would make a difference. In this section, we discuss how to do it and present a prototype to check temporal properties under this interpretation. There is a theoretical advantage in that any ω-regular strategy on a Maude model can be checked now, and a practical one in that fairness constraints of the model can be expressed as part of the strategy itself. However, this change has some cost on efficiency.

Standard model-checking algorithms assume that models are plain transition systems that cannot handle restrictions on the whole infinite execution, like those imposed by acceptance conditions on ω-automata. These properties are usually expressed as premises in the formulae to be verified or given as another input to the model-checking algorithms. Sometimes it is more natural to think about these restrictions as part of the system specification, and so fair Kripke structures have been proposed [ASB⁺94] with the same conditions as ω-automata. The automata-based LTL model-checking algorithm can handle this enhanced models with small changes, since the model is represented as a Büchi automaton too.

In order to forbid infinite iterations on strategy expressions, we will establish some acceptance conditions for the reachable iterations of the model. Out of the condition classes seen in

[10] Notice that μ-calculus properties that refer to both edge and state labels cannot be verified with the last LTSmin version at the moment of writing (3.0.2). We have proposed a change that makes it possible and the modified version is available for download in [EMM⁺21] in the meantime.

Section 2.1, the intuition is that a co-Büchi condition including all states or transitions where an iteration starts would be fine, since this would ensure that they are only visited finitely many times. However, iterations could be executed infinitely many non-consecutive times within nonterminating recursive calls specifying infinite executions. For example, a recursive strategy `mt` like `mt := rl * ; mt` must be able to execute infinitely many times `rl` but only finitely many times in each strategy call. This legitimate possibility complicates the definition of the acceptance conditions and forces to consider each distinct iteration separately, for what we introduce the following technical definitions.

Definition 4.5. *For any $\pi \in \mathcal{X}S^\omega$ and set of variables X, we define the following:*

- *A* position *in an execution state is a word $p \in X^*$. The substate of q at position p is written $q|_p$ and defined by $q|_\varepsilon = q$ and $(\mathrm{subterm}(\ldots, x : q, \ldots ; t) \,@\, s)|_{xp} = q|_p$.*
- *A* partial context *is a word $c \in (\mathrm{Strat} \cup \mathrm{VEnv} \cup X)^*$, and it is contained in an execution state $c \in q$ if $q = t \,@\, c$ for some term t, or $c = c_1 x c_2$ for $x \in X$, $q = \mathrm{subterm}(\ldots, x : q', \ldots ; t) \,@\, c_2$ and $c_1 \in q'$. A partial context is contained in an execution $c \in \pi$ if $c \in \pi_n$ for some $n \in \mathbb{N}$.*
- *The position of a partial context* $\mathrm{pos}(c)$ *is defined as* $\mathrm{pos}(\alpha s) = \mathrm{pos}(s)$, $\mathrm{pos}(\sigma s) = \mathrm{pos}(s)$, $\mathrm{pos}(x s) = x\,\mathrm{pos}(s)$, *and* $\mathrm{pos}(\varepsilon) = \varepsilon$.
- *A partial context is a strict suffix of another $c \sqsubsetneq_s c'$ if the word c is a strict suffix of c'.*
- $\mathrm{Enter}(c, \pi) := \{n \in \mathbb{N} : \pi_n \rightarrow_c^* q \rightarrow_c q' \twoheadrightarrow \pi_{n+1} \;\wedge\; c \in q \;\wedge\; q'|_{\mathrm{pos}(c)} = t \,@\, c' \;\wedge\; c \sqsubsetneq_s c'\}$.
- $\mathrm{Leave}(c, \pi) := \{n \in \mathbb{N} : \pi_n \rightarrow_c^* q \rightarrow_c q' \twoheadrightarrow \pi_{n+1} \;\wedge\; c \in q \;\wedge\; q'|_{\mathrm{pos}(c)} = t \,@\, c' \;\wedge\; c' \sqsubsetneq_s c\}$.

The free variables in these sets are quantified existentially.

Two iterations are considered the same if their partial contexts coincide, and they take place in fixed positions of the execution state. The previous definition ignores `rewc` states, since infinite iterations within them will never produce $\twoheadrightarrow$ executions and so rewriting paths, because steps in rewc are always control transitions. The final two sets can be used to describe the indices where an iteration rule is executed: to start another round of the iteration $\alpha\!*\, c$ in $\mathrm{Enter}(\alpha\!*\, c, \pi)$, or to leave the same iteration with $\mathrm{Leave}(\alpha\!*\, c, \pi)$, depending on whether the stack grows or shrinks. Clearly, an iteration is not executed forever if its Enter set is finite or its Leave set is infinite, because the first contains an element for each iteration round and the steps in the second interrupt them. Recalling that the reason why an iteration may enter infinitely many times without iterating forever are recursive calls, the second set can be replaced by the Enter of the largest suffix of the partial context of the iteration with a recursive call context on top.

Proposition 4.8. *Given an execution π of the semantics, π iterates finitely iff for every partial context $\alpha^* c \in \pi$,* $\mathrm{Enter}(\alpha\!*\, c, \pi)$ *is finite or* $\mathrm{Leave}(\alpha\!*\, c, \pi)$ *is not finite. Moreover, this happens iff* $\mathrm{Enter}(\alpha\!*\, c, \pi)$ *is finite or* $\mathrm{Enter}(c', \pi)$ *is infinite, where c' is the longest suffix of c whose top element is the context of a recursive call or ε if there is none.*

The previous Enter and Leave sets can be used to define Streett acceptance conditions for the model automaton. Consider the graph of the semantics from an execution state $t \,@\, \alpha$; assuming it is finite, finitely many iterations $C = \{\alpha_1\!*\, c_1, \ldots, \alpha_n\!*\, c_n\}$ must have been executed to generate it. The conditions in Enter and Leave for the transitions of an execution can also be evaluated on the transitions of the graph. For the first statement of Proposition 4.8, if A_k are the Enter transitions for $\alpha_k\!*\, c_k$ and B_k are the Leave transitions, $\{(A_k, B_k) : 1 \leq k \leq n\}$ is a Streett condition on the transitions of the model ensuring that iterations are not executed forever. However, acceptance conditions are usually attached to the states, so A_k and B_k should be the sets of states where these transitions end. States where different iterations start or end may need to be duplicated. The same can be done for the Enter-Enter pairs of the second statement of the proposition, with the advantage that the sets of two iterations under the same recursive call can be safely merged. In case there are no recursive calls, a single co-Büchi condition would be obtained, as mentioned at the beginning.

Proposition 4.9. *Given a strategy α and an initial term t, the Streett automaton produced from $\mathcal{M}_{\alpha,t}$ as indicated above accepts the language $E_K(\alpha, t)$.*

Streett conditions can be translated to Büchi conditions by paying a blowup of $n(1 + k2^k)$ in the worst case where n is the number of states of the original automaton, and k the number of Streett pairs [CHV⁺18; §4.3.1]. Moreover, since the system automaton does no longer have trivial Büchi conditions, intersecting it with the property automaton is harder than in the usual case and the number of states of the product automaton may double. Hence, simplifying and reducing the number of Streett pairs is always convenient.

Alternatively, we can label states where iterations take place with atomic propositions and discharge the fairness restriction to the property logic. The following proposition proves that it is generally possible with LTL, and similarly for other logics.

Proposition 4.10. *Given a Maude strategy expression α, an LTL property φ, and assuming that the iterations that occur in the reachable states from $t \mathbin{@} \alpha$ are numbered from 1 to n, $(\mathcal{M}, E_K(\alpha)) \vDash \varphi$ is equivalent to $\mathcal{M}^K_{\alpha,t} \vDash \psi \to \varphi$ where $\mathcal{M}^K_{\alpha,t} := (\mathcal{XS} \times \mathcal{P}(AP'), \twoheadrightarrow', \{t \mathbin{@} \alpha\}, AP \cup AP', \ell')$ where $AP' = \{e_1, l_1, \ldots, e_n, l_n\}$, $\ell'((q, U)) = \ell(q) \cup U$, $(q, U) \twoheadrightarrow' (q', U')$ if $q \twoheadrightarrow_{\mathrm{Sol}} q'$, and U' contains e_k or l_k if the iteration k has been entered or left from q to q' (as explained above), $\psi = \bigwedge_{k=1}^n \psi_k$, and $\psi_k = \Diamond\Box\neg e_k \ \vee \ \Box\Diamond l_k$.*

The finite-iteration semantics can also be respected in CTL* by adding the restriction ψ as premise to all path quantifiers, i.e. replacing ϕ in each $\mathbf{A}\,\phi$ by $\psi \to \phi$ and in each $\mathbf{E}\,\phi$ by $\psi \wedge \phi$. In particular, CTL properties can be transformed likewise yielding CTL* properties, and the same effect can be achieved with the fairness constraints supported by some model checkers [CHV⁺18]. However, the previous proposition requires duplicating states in some corner cases and adding atomic propositions whose satisfaction does not depend exclusively on the subject term, so users of the model checker in Section 4.3 would not be able to do it. Moreover, states should be duplicated carefully not to introduce the problems seen in Section 4.2.1. This approach is implemented in the following prototype that gets track of iterations.

4.5.1 A model checker prototype for Kleene-star iterations

The procedures proposed in Propositions 4.8 and 4.10 above are implemented within `umaudemc` as a prototype model checker. Since the strategy-controlled rewrite graph does not trace the execution of iterations, an executable Maude specification of the small-step operational semantics described in Section 7.3 is used to generate the model. Transformed formulae with the fairness premises of Proposition 4.10 can be used to check LTL, CTL, and CTL* properties on this system under the new semantics. States should be merged as usual regardless of the iteration annotations. Alternatively, for LTL properties and using the Spot library, a Streett automaton is built for the model with the conditions mentioned in Proposition 4.8, obtained from the graph of the executable semantics. As usual in the automata-theoretic approach, another Büchi automaton is generated by Spot for the negation of the LTL formula, and an accepting run is searched in their intersection. However, the state space is expanded eagerly, because on-the-fly algorithms are not used when the acceptance condition contains finiteness constraints. Moreover, this expansion is also required to compute the acceptance sets, and unavoidable when using the Python interface to Spot, since on-the-fly model checking is only supported through C++. This new model checker is available through the `check` subcommand of the `umaudemc` interface, where the flag `--kleene-iteration` enables this interpretation.

Let us illustrate the new model checker with a simple example. The cups and balls is an ancient performance made with a table, three cups, and a small ball in one of its multiple variations. The illusionist puts the three cups upside down on the table, covering the marble with one of them, then swaps them randomly multiple times, and asks the audience to guess in which cup the ball is inside. The selected cup is raised to show whether the guess was right.

```
mod CUPS-BALLS is
  protecting LIST{Nat} .

  sorts MaybeBall Cup Table .
  subsorts MaybeBall Cup < Table .

  ops ball nothing : → MaybeBall [ctor] .
  op  cup          : MaybeBall → Cup [ctor frozen] .
  op  empty        : → Table [ctor] .
  op  __           : Table Table → Table [ctor assoc id: empty] .

  var  T          : Table .
  vars B? B1? B2? : MaybeBall .

  rl [swap]    : cup(B1?) T cup(B2?) ⇒ cup(B2?) T cup(B1?) .
  rl [uncover] : cup(B?) ⇒ B? .
  rl [cover]   : B? ⇒ cup(B?) .

  op initial : → Table .
  eq initial = nothing ball nothing .
endm
```

The table is a list mixing `cups`, `balls`, and `nothing` placeholders, and the three basic actions of the illusionist are specified as the rules `swap`, `uncover`, and `cover`. The initial position contains a ball in the middle of the table and nothing elsewhere, and the continuous repetition of the spectacle is specified by the following recursive strategy, where swapping is executed by an iteration.

```
smod CUPS-BALLS-STRAT is
  protecting CUPS-BALLS .

  strats cups cups-rec @ Table .

  sd cups := cover ! ; cups-rec .
  sd cups-rec := swap * ; uncover ; cover ; cups-rec .
endsm
```

Two atomic propositions `uncovered` and `hit` are defined to label the tables where the content of a cup is uncovered and where the ball is visible. As usual, we also define the module `CUPS-BALLS-CHECK` where the complete specification and the `STRATEGY-MODEL-CHECKER` module are combined.

```
mod CUPS-BALLS-PREDS is
  protecting CUPS-BALLS .
  including SATISFACTION .

  subsort Table < State .
  ops uncovered hit : → Prop [ctor] .

  vars L R T : Table .     var B? : MaybeBall .

  eq L B? R   ⊨ uncovered = true .
  eq T         ⊨ uncovered = false [owise] .
  eq L ball R ⊨ hit = true .
  eq T         ⊨ hit = false [owise] .
endm
```

The property $\Box\Diamond$ *uncovered* should be satisfied in this system controlled by the `cups` strategy when the iteration is understood as the Kleene star. However, in the normal interpretation of the main model checker, the iteration of `swap` applications can last forever and the property is not satisfied.

```
$ umaudemc check cupsballs.maude initial '[] <> uncovered' cups --backend spot
The property is not satisfied in the initial state
(22 system states, 71 rewrites, 2 Büchi states)
| nothing ball nothing
v  cover
| cup(nothing) ball nothing
v  cover
| cup(nothing) cup(ball) nothing
v  cover
| cup(nothing) cup(ball) cup(nothing)
v  swap
| | cup(nothing) cup(nothing) cup(ball)
| v  swap
< v
```

If we add the `--kleene-iteration` or `-k` flag to the command, the model checker in this section is used instead, showing the expected result.

```
$ umaudemc check cupsballs.maude initial '[] <> uncovered' cups -k
The property is satisfied in the initial state
(48 system states, 12517 rewrites, 2 Büchi states)
```

In this case, two distinct iterations have been found, whose partial contexts are `cover * ; not(cover) ; cups-rec` for the `!` operator, and `swap * ; uncover ; cover ; cups-rec` for the explicit one. If Spot is available, the automata intersection procedure is followed by default, like in the execution above. Otherwise, the formula is transformed and checked on the small-step semantics with another model-checking backend. Using the Maude LTL model checker, the previous property is verified by the second method almost as fast as by the first, even though the Büchi automaton has 22 states. CTL* properties can also be checked by the latter method, but the complexity of the transformed formulae makes it impractical for large examples. The property $\mathbf{A}\,\Box\,\mathbf{E}\,\Diamond\,hit$ says that it is always possible to get the guess right, and it is translated to $\mathbf{A}\,(\Diamond\Box\neg e \lor \Box\Diamond\, l \to \Box(\mathbf{E}\,((\Diamond\Box\neg e \lor \Box\Diamond\, l) \land \Diamond\, hit)))$ for the following problem, where the initial covering of the objects has been skipped.[11]

```
$ umaudemc check cupsballs.maude 'cup(nothing) cup(ball) cup(nothing)'
  'A [] E <> hit' cups-rec -k --backend pymc
The property is satisfied in the initial state
(27 system states, 6846 rewrites, holds in 27/27 states)
```

However, illusionists and conmen sometimes make the ball disappear to obtain the admiration or the money of the public. Adding this possibility to our rules and strategies is a small change.

```
  rl [disappear] : cup(ball) ⇒ cup(nothing) .
  rl [appear]    : cup(nothing) ⇒ cup(ball) .
```

[11]States are not properly merged with the LTSmin backend since its plugin does not directly handle the extended states of the executable semantics. Hence, the less efficient `pyModelChecking` backend is used to check the branching-time properties in this section.

```
sd cups2 := cover ! ; cups2-rec .
sd cups2-rec := (swap | disappear) * ; not(amatch cup(call)) ;
    uncover ; cover ; appear ; cups2-rec .
```

According to the `cups2-rec` definition, the ball can be distracted at any time while swapping the cups with the `disappear` rule, and the test ensures that this is actually done. Then the property $\mathbf{A}\,\square\,\mathbf{E}\,\diamond\, \mathit{hit}$ is no longer satisfied.

```
$ umaudemc check cupsballs.maude 'cup(nothing) cup(ball) cup(nothing)'
  'A [] E <> hit' cups2-rec -k --backend pymc
The property is not satisfied in the initial state
(15 system states, 9739 rewrites, holds in 0/15 states)
```

Another application of this model checker is shown in Section 6.3, where the meaning of the iteration ensures that some processes in a processor release it infinitely often.

4.6 The satisfiability problem

The study of a model-checking problem is usually accompanied of the study of its satisfaction problem. In our case, fixed a Kripke $\mathcal{K}$ structure and a temporal property φ, its formulation can be the following question: is there any strategy E such that $(\mathcal{K}, E) \vDash \varphi$? If the model is given by a rewriting system in Maude, is there any expression α such that $(\mathcal{K}, E(\alpha)) \vDash \varphi$? Variations of this problem are related to the general topic of program synthesis, supervisory control [RW84], reactive synthesis [PR89], etc. However, the latter are posed for *open systems* where the synthesized controller must ensure the satisfaction of certain properties in a model also affected by an uncontrollable environment. Our problem is easier, since the strategy will have full control on the whole system.

When considering linear-time properties, this question has a trivial answer unless other restrictions are imposed. In fact, the empty strategy $\varnothing$ (or **fail** in the Maude strategy language) will satisfy any property. If the strategy is only required to be non-trivial, satisfaction is reduced to model checking the negated property, since this will either confirm its unsatisfiability or provide a counterexample w such that $\{w\}$ is a strategy for which the property is satisfied. Furthermore, if we wish to find the most general strategy that makes the system satisfy the property, $E = \{\pi \in \Gamma_\mathcal{K} : \mathcal{K}, \ell(\pi) \vDash \pi\}$, we would only have to calculate the intersection between the executions allowed by φ and the executions of the system $\Gamma_\mathcal{K}$. In the case of LTL, whose $L(\varphi)$ is an ω-regular language, this is always possible with the same procedure used for model checking, but with the automaton for φ instead of $\neg\,\varphi$.

Proposition 4.11. *Given a Kripke structure $\mathcal{K}$ and an LTL formula φ, $E = \{\pi \in \Gamma_\mathcal{K} : \mathcal{K}, \ell(\pi) \vDash \varphi\}$ is an ω-regular strategy and its calculation is* PSPACE*-complete in the size of $\mathcal{K}$ and the Büchi automaton of φ.*

In the case of the Maude strategy language, it is possible to find a strategy expression α that denotes the most general strategy as long as this intersection is recursively enumerable by Proposition 3.2. However, Propositions 3.3 and 3.4 imply that only for a closed ω-regular intersection the strategy α will have a finite execution space and be model checkable in practice. Non-closed ω-regular intersections can also be handled if the iteration of the strategy language is interpreted as the Kleene star like in Section 3.6.1 using the model checker in Section 4.5. Alternatively, when using this semantics and for LTL properties, the intersection need not be calculated and instead we can construct a strategy for the ω-regular expression of $L(\varphi)$ as in the proof of Proposition 3.4, translating the base symbols $U \subseteq AP$ to strategy expressions **match** P **s.t.** $\bigwedge_{u \in U} P \vDash u$; **all**.

For branching-time properties, satisfiability is usually much harder than model checking. For example, the complexity of CTL* model checking is PSPACE-complete while its satisfaction problem is 2EXPTIME-complete [VS85]. As anticipated in Chapter 2, strategic logics like

ATL* [AHK02] and SL [MMP⁺14] would only apparently subsume our satisfaction problem for CTL* as a very particular case. While the semantics of strategy quantification and the type of strategies considered in the most used variants of these logics impede establishing this relation, there are some that support nondeterministic strategies and match the configuration of our problem, namely Updatable Strategy Logic (USL) [CBC15] and SL$^{\prec}$ [GMM20]. Since the operators of SL$^{\prec}$ are closer to those of CTL*, we will see how this logic solves its satisfaction problem with strategies.

Remember from Section 2.6 that the models of SL$^{\prec}$ are concurrent game structures (CGS) $G = (S, Ag, Ac, \tau, s_0, AP, \ell)$, whose transition function $\tau : S \times (Ag \to Ac) \to S$ maps a state and the combined decision of the agents to the next state. Formulae in SL$^{\prec}$ are built with the temporal operators of CTL* plus a strategy quantifier $\exists x \;\; \varphi$ that is satisfied if there is a nondeterministic strategy $S^+ \to \mathcal{P}(Ac)$ such that φ holds, a strategy binder (a, x) that binds the agent a to the strategy x, and other operators that will not be relevant here. The path quantifiers $\mathbf{E}$ and $\mathbf{A}$ range over the paths yielded by the executions of the game where players obey their strategies.

Given a Kripke structure $\mathcal{K} = (S, \to, \{s_0\}, AP, \ell)$, we consider a one-player CGS

$$G = (S \cup \{\bot\}, \{sys\}, S, \tau, s_0, AP \cup \{p_\bot\}, \ell_\bot)$$

where the single player is called *sys*, its actions are the states of the model, and $\ell_\bot(s) = \ell(s)$ for $s \in S$ and $\ell_\bot(\bot) = \{p_\bot\}$. For any decision $c : \{sys\} \to S$ of the only player, the transition function is defined by $\tau(\bot, c) = \bot$, $\tau(s, c) = c(sys)$ if $s \to c(sys)$, and $\tau(s, c) = \bot$ otherwise, so that only the moves allowed by $\to$ are done and bad choices get trapped in the state $\bot$.[12] Then, for any CTL* property φ, consider the formula $\psi \coloneqq \exists x \; (sys, x) \; \mathbf{A} \square \neg p_\bot \wedge \varphi$. This property holds iff there is a strategy $\lambda : S^+ \to \mathcal{P}(S)$ such that $\mathbf{A} \square \neg p_\bot \wedge \varphi$ holds when *sys* runs λ. The first clause indicates that all movements are allowed by the transition relation of $\mathcal{K}$ and the second clause is the desired property, so this holds iff there is an intensional strategy $\lambda : S^+ \to \mathcal{P}(S)$ such that $(\mathcal{K}, E(\lambda)) \vDash \varphi$, solving our satisfaction problem. The complexity of SL$^{\prec}$ model checking is $(k+1)$-EXPTIME-complete depending on the simulation depth k of the formula [GMM20], which is at most 1 for ψ. Hence, the complexity of the strategy-controlled satisfaction for CTL* is in 2EXPTIME.

4.7 Implementation

The essential part of the implementation of the different model-checking tools in this thesis is the generation of the rewriting system controlled by an expression in the Maude strategy language, which relies on C++ infrastructures of the Maude LTL model checker and of the strategy execution engine, explained in Section 3.7. As mentioned in Section 4.3, the modular design of the Maude LTL model checker allows reusing most of its functionality except the on-the-fly generator of the state and transition graph of the model. The model is exposed as a C++ object with the following signature:

```
struct System {
  virtual int getNextState(int stateNr, int transitionNr) = 0;
  virtual bool checkProposition(int stateNr, int propositionIndex) const = 0;
};
```

States are indexed by natural numbers, and their successors are queried using the `getNextState` method with increasing values of `transitionNr` until it returns -1. Atomic propositions are evaluated on states via the `checkProposition` method, and the values of both functions can be calculated on the fly as they are requested. In both the original and the strategy-aware model checker, states are univocally associated to a term t and atomic propositions are terms p as

[12] Other definitions of CGS include a function $d : Ag \times S \to \mathcal{P}(Ac)$ that limit the available actions for each agent at each state, which will make the state $\bot$ and $p_\bot$ unnecesary, but we want to stick to the CGS definition used for SL$^{\prec}$.

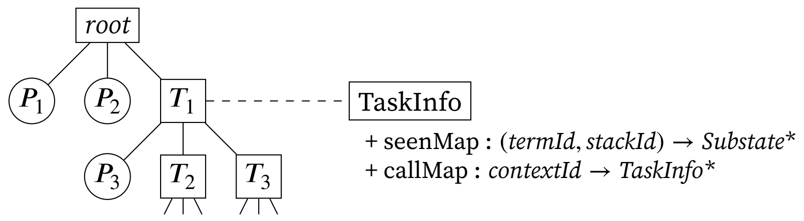

root
P_1
P_2
T_1
P_3
T_2
T_3
TaskInfo
+ seenMap : (*termId*, *stackId*) → *Substate**
+ callMap : *contextId* → *TaskInfo**

Figure 4.8: Example hierarchy of tasks (boxes) and processes (circles) with a task info.

specified in Section 4.3, so `checkProposition` evaluates them by reducing $t \models p$. While the states of the original model consist merely of this term t, those of the strategy-aware model also incorporate the strategy execution state.

The successors of a state are calculated using the infrastructure of the commands `srewrite` and `dsrewrite`. This is supported by a collection of tasks and processes that have been slightly and conveniently adapted. Remembering the more detailed description of Section 3.7, processes are in charge of applying rules and advancing the strategy execution, for what they may destroy and create new processes and tasks, and all of them are maintained in a global doubly-linked list and executed in a round-robin or FIFO policy depending on the command. Each process is attached to a task, which are organized hierarchically as a tree, as illustrated in Figure 4.8. Tasks group processes being responsible for the same subsearch and delimit the variable environments produced by the **`matchrew`** operator or strategy calls. Moreover, each task maintains a set of visited term-strategy pairs to avoid repeating unnecessary calculations and to let the search terminate in the presence of cyclic executions. The visited set of each task is independent, because the same strategy could be applied to the same term but with other values for the variables or in a different subsearch. The pending strategies are handled by a queue similar to those of the operational semantics, and in fact the strategies of the term-strategy pairs are indices to this structure. Each task also holds the index of the pending strategies to be executed for each solution of the subsearch it hosts.

For the model checker, small but essential changes are applied to this infrastructure. First, the global list of processes is split into multiple lists local to each model state to allow calculating and identifying their successors. Each model state stores a pointer to the current process in its list, which is executed in round-robin. According to the $\twoheadrightarrow$ semantics, a new state is only generated when a rewrite takes place (or an opaque strategy yields a result, see Section 4.3.1), when the active process notifies it to the object in charge of managing the model graph. At that moment, checking whether the new state has been visited before is crucial to ensure the termination of the algorithm in the conditions indicated in Section 4.3, and doing it safely and efficiently is perhaps the most complicated aspect of the implementation. Ideally, two model states are equivalent if they correspond to the same execution state of the operational semantics. Checking the equivalence just at the state creation is enough not to lose any cycle, but actually, a model-checker state visits many states of the semantics, always related by control transitions, when executing its list of processes. Some of them may be as general as the formal state represented by the initial process, having the same successors by the $\twoheadrightarrow$ transition, but others may have lost continuations because of a rule like $\alpha \,|\, \beta \to_c \alpha$. In order to anticipate the detection of cycles, with the consequent advantages in execution time and simplicity of the possible counterexamples, the model controller executes as many deterministic operations as possible to compare with a simpler instance of the state, it generates *substates*[13] to reuse when convenient the search from the alternative branches that grow from the states, and merges states if their equivalence is detected afterwards. The aliasing of system states produced by this last improvement is reflected in the second number of states printed in the verbose mode.

[13]Substates are entirely similar to states except that they are not part of the model, and consequently they are not linked as successors by other (sub)states, but as *dependencies*, from which successors are copied instead.

```
Maude> red modelCheck(initial, [] ~ risky, 'eagerEating2) .
ModelChecker: Property automaton has 2 states.
StrategyModelCheckerSymbol: Examined 5 system states (4 real).
rewrites: 15
result ModelCheckResult: counterexample(...)
```

The correspondence from an implementation state to a state of the semantics is based on adding to the subject term t being rewritten by the current process the pending strategies according to the strategy stack index $t \mathbin{@} \overline{\alpha}$, the variable environment and continuation of the enclosing parent task $t \mathbin{@} \overline{\alpha}\theta\overline{\beta}$, and the appropriate $\mathcal{XS}$ constructor like $\mathrm{subterm}(\ldots, x_i : t \mathbin{@} \overline{\alpha}, \ldots)$ according to the parent task too. Hence, checking if two model states are equivalent goes through comparing their subject terms, their pending strategy indices, and their ancestor tasks. The first two were already compared in the normal execution using the task-local visited sets, but this is insufficient for several reasons. On the one hand, aborting the search when detecting a visited state is not an option here because we must know how the execution continued to complete the graph, so the visited set should be replaced by a table. On the other hand, as per the tail-recursive call optimization described in Section 3.5, the model can be finite even in the presence of nonterminating strategy calls if these are tail recursive with finitely many different arguments. The execution infrastructure does not compare the arguments of the strategy calls and generates a different task for each call, and so this circumstance is not detected. Both problems are solved associating to each task a *task info* structure (see Figure 4.8) holding the aforementioned table, which maps each term-strategy pair to the substate that continues its execution, and another table associating variable environments to the *task info* structure shared by all the recursive strategy call tasks starting there. Except for this case, the state comparison is done locally at the task level and this may delay the detection of cycles in some cases. For example, if β is **matchrew** x **by** x **using** r and r is a rule that rewrites a to b and b to a, a cycle like

$$\begin{aligned}\mathrm{subterm}(x : \mathtt{a} \mathbin{@} \mathtt{r}; x) \mathbin{@} \beta^* \twoheadrightarrow \mathrm{subterm}(x : \mathtt{b} \mathbin{@} \varepsilon, x) \mathbin{@} \beta^* &\to_c^* \mathtt{b} \mathbin{@} \beta^* \\ &\twoheadrightarrow \mathrm{subterm}(x : \mathtt{a} \mathbin{@} \mathtt{r}; x) \mathbin{@} \beta^*\end{aligned}$$

will not be detected in its final state. The reason is that the tasks for the first and last subterms are not the same: the first has been destroyed when the execution of the **matchrew** has finished and the second is a new one with a fresh table of visited pairs. However, no cycle will be missed in this situation or a similar one, because the execution must evolve to a lower level in the task hierarchy, in this case to $\mathtt{b} \mathbin{@} \beta^*$, when the parent task will be the same and the cycle will be detected. Not to miss any such case, the visited table is always looked up when an execution descends to a parent task. Obviously, a deeper comparison of the tasks could prevent this inconvenience at a higher cost. A compromise should be found between state-space reduction, speed and memory required for each state, always ensuring that the algorithm finishes when the abstract execution states are finite.

All things considered, the model-checker states represented in C++ correspond to states of the operational semantics, and exactly those reachable from the initial state $t \mathbin{@} \alpha$ by the $\twoheadrightarrow \; = \; \to_c^* \circ\!\!\!\twoheadrightarrow \to_s$ transition. The formal states represented by the first and second state of a model checker's transition are linked by a $\twoheadrightarrow$ step, and the formal states of all substates on which they depend are linked by $\to_c^*$ transitions. The only exceptions are opaque strategies, where connected states are linked by the aggregated $\to_{s,c}^*$ step that completes the strategy execution, and solution states, where safe self-loops are added to extend finite traces to infinite ones. The cycle detection mechanism ensures that the algorithm terminates under the assumptions of Section 4.2.

4.7.1 External model checkers

The Kripke-like interface of both the classical and the strategy-controlled models just explained can be accommodated to the *gray box* interface of the LTSmin toolset and to construct specifications for other verification tools. As explained in Section 4.4, the Maude plugin for LTSmin essentially acts as an adapter from the PINS interface to the Maude interface, and `umaudemc` and the Python `maude` library bring access to these models and useful helpers for building the input data of other tools or programming other external checkers. However, properly checking branching-time properties requires implementing the adaptations explained in Section 4.2.1: purging failed execution states that cannot be continued to a valid execution, and merging successors with a common underlying term. The first tweak is done before the model-checking algorithm actually starts and expands the full state space. This is not a serious drawback, since branching-time algorithms will expand the full state space in any case, and it is almost unavoidable because checking whether a single execution state is valid may still require expanding an arbitrary number of states. Algorithm 1 implements the purge of failed states with a complexity of $\Theta(n + m)$ where n is the number of states and m the number of transitions, and produces a Boolean vector marking the subset of valid states. It is similar to the Tarjan's strongly connected component algorithm [Tar72]: during a depth-first search every pending state is annotated as if it were valid, because if it is ever reached while exploring its successors, a cycle containing it will have been found, and so it is actually valid. However, if no valid successors are found after the recursive search, the state is marked as failed. Solution states should be considered valid in any case, although this follows from their self-loops.

```
Input: A graph $G = (V, E)$ whose vertices are integers and whose transitions are
       obtained with "getNextState".
Output: A Boolean vector "validStates" describing $U \subseteq V$ such that for all $u \in U$ there
        exists a $u' \in U$ with $u \to^* u'$ and isSolution($u'$) or $u' \to^* u'$.
Function Expand(int state) is
    index = 0;
    stateIsValid = false ;
    nextState = getNextState(state, 0);
    while nextState $\neq$ -1 do
        if nextState $\geq$ length(validStates) then
            validStates.append(true) ;
            Expand (nextState) ;
        end
        if validStates[nextState] then
            stateIsValid = true;
        end
        nextState = getNextState(state, ++index);
    end
    validStates[state] = stateIsValid;
end
```

Algorithm 1: Failed state removal algorithm.

The adaptation that merges successors with a common term projection is summarized in Algorithm 2. A local table is used to group the successors of all the execution states in the merged-state set, and these groups are communicated as successors to the caller. A global table is kept to index and look for the states that have already been visited. For labeled transition systems, the only change is that the table should be indexed by both term and transition label.

These two algorithms are implemented in C++ as part of the LTSmin plugin, and in Python to serve all backends directly supported by `umaudemc`. Iterative instead of recursive algorithms are used in Python due to its tight limitation on the number of recursive calls. Whether these adaptations are used or not is chosen depending on the input formula by `umaudemc`, and failed states are removed before applying Algorithm 2. For the tableau-based LTL implementations of NuSMV and `pyModelChecking`, failed states are removed but they are not merged.

For the external model checkers, states are integer indices that may indistinctly refer to the internal Maude rewriting graph or to the table of merged states. These numbers are assigned

```
Data: A global vector “states” of merged states (sets of indices to execution states of the
      Maude internal model) and its reverse lookup table “globalTable“.
Function getNextMergedState(int mergedState) is
    Create an empty “table” from terms to sets of state indices ;
    foreach state ∈ states[mergedState] do          // (1) grouping successors by state
        index = 0 ;
        nextState = getNextState(state, 0);
        while nextState ≠ -1 do
            term = getStateTerm(nextState) ;
            if term ∈ table then
                table[term].add(nextState) ;
            else
                table[term] = {nextState} ;
            end
            nextState = getNextState(state, ++index);
        end
    end
    foreach term ↦ group ∈ table do                 // (2) producing merged successors
        if group ∈ globalTable then
            index = globalTable[group] ;
        else
            index = length(states) ;
            states.append(group) ;
            globalTable[group] = index ;
        end
        notify_successor_to_caller(index, ...) ;
    end
end
```

Algorithm 2: Merging successor states algorithm.

on the fly depending on how the state space is explored and do not have any persistent meaning. As a result, the concurrent `pins2lts-mc` and distributed `pins2lts-dist` tools of LTSmin cannot be used in general with our language module, since they interchange state indices that may have a different meaning or no meaning at all in a different instance. However, thread-level parallelism as implemented by the `pins2lts-sym` tool is compatible with our plugin, although limited by the fact that the basic Maude operations like reducing and rewriting are not reentrant, and they must be protected by a critical section. The translation of μ-calculus formulae in `umaudemc` for LTSmin is tricky, since there are two different syntaxes handled by different implementations: `mu` that does not support edge label specifications but only the `<.>` and `[.]` variants, which are written `E` and `A`; and `mucalc` that admits edge labels but does not support state labels without a modification that we have implemented and proposed to their authors. Hence, the `umaudemc` tool translates μ-calculus formulae without edge labels to the `mu` syntax and invokes the symbolic tool `pins2lts-sym` to check them, and otherwise uses the `mucalc` syntax and invokes the sequential tool `pins2lts-seq` with our modifications, and then the `pbespgsolve` parity game solver from mCRL2 to obtain the verification result.[14]

Historically, our first approach to check branching-time properties was producing a NuSMV source file via a Maude program that used the metalevel functions `metaXApply` and `metaReduce` to construct the transition graph. However, this cannot be easily extended to systems controlled by strategies, since the current Maude metalevel does not provide enough functions to inspect and follow their executions.

4.8 Related work

In Chapter 2 we have reviewed other strategy languages used in formal specification, other extensions of model checking in the context of Maude, and other uses of strategies in formal verification of multiagent systems and games. In this section, we focus on their similarities and differences regarding the model-checking problem addressed in this thesis.

Strategic logics fruitfully relate strategies and model checking. However, strategies are not part of the model specification like in this work, but instead they are a dialectic resource of the language in which properties are expressed. Sentences in the different strategy logics reason about strategies without describing them, since they are always introduced by existential and universal quantification. Consequently, our model-checking problem cannot be directly seen as a particular case of the problem for these logics, although our satisfaction problem can be solved as a very particular case of the model-checking problem for some of these logics, as seen in Section 4.6. However, the satisfaction of their formulae is defined recursively after quantifiers have formally assigned a strategy to an agent or variable, so they indirectly define the satisfaction of temporal properties in strategy-controlled systems. The evaluation of φ and $(a, x)\,\varphi$ under an assignment that maps x or a to an intensional strategy $\lambda : S^+ \to \mathcal{P}(Ac)$ coincides with our concept of model checking in Definition 4.3, or more precisely, to the generalized CTL* semantics of Section 4.1.1. Most strategic logics are not defined in the general context of nondeterministic strategies we are considering, although some of them are, like USL [CBC15] and SL$^{\prec}$ [GMM20]. As mentioned in Section 2.6, there are model checkers for subsets of more restrictive strategy logics that represent strategies explicitly as witnesses of the satisfaction of a property, as the output of the model checker instead of as its input.

This possibility is also available in a different context using Uppaal Stratego [DJL$^+$15] from the Uppaal [LYP$^+$] modeling environment for real-time systems. Moreover, the synthesized strategies can be later used to execute the constrained system and model check it against other properties, considering only this restricted "strategy space". Strategies in this case are memoryless and deterministic, but they follow the same idea of this work.

[14]The symbolic tool `pins2lts-sym` also supports `mucalc` formulae and even includes a builtin parity game solver to complete the verification. However, we have found a bug in its implementation that we have reported to the authors.

Something similar to strategies is provided as input to the model-checking algorithms in other contexts. In *propositional dynamic logic* (PDL) [FL79] and *linear dynamic logic* (LDL) [GV13], formulae $\langle r \rangle \varphi$ include complex actions r built on top of edge labels using regular expression combinators and tests, which can be seen as a subset of the Maude strategy language. Atomic propositions and more complex temporal formulae can be checked at the end of those sequences of actions or at arbitrary points during them using tests. Unlike in our approach, properties are not checked in the system restricted by the action patterns, but at some execution points indicated by them.

Strategies have also been applied to reduce the search space for the sole purpose of model checking. Like search heuristics [Pea84], they can be used to guide the model checker search to a counterexample or witness of the desired property [ELL04, GV04]. For directing symbolic reachability problems, a language of reachability expressions [TCP08] was proposed including union, concatenation, and iteration operators that resemble those of Maude and similar strategy languages. These applications are also possible with our model checker, and an example of how restricting the search may yield simpler counterexamples is shown in Section 6.3.

In the context of rewriting logic, an alternative to control rewriting can be found in the compositional specification approach of [MVM20]. Systems are there described as a collection of components synchronized on arbitrary user-defined properties of their states and transitions, which are considered on equal terms. One of these components can play the role of a strategy or controller that limits the behavior of the rest of the composed system. A synchronous product of the components yields a plain Maude specification where temporal properties can be checked with the standard Maude model checker. Compositional verification techniques are under development.

Chapter 5

A probabilistic extension of Maude and its strategies

In this chapter, we temporarily abandon the classical setting of nondeterministic transition systems and qualitative properties in which the rest of the thesis is focused. Probabilistic models and quantitative verification are discussed in the context of Maude and its strategy language. There are some precedents of the integration of probabilistic features into rewriting logic and Maude we have already mentioned in Section 2.7. PMaude [AMS06] is an extension of Maude for specifying probabilistic rewrite theories with rules that may contain additional variables in the righthand side to be instantiated by sampling probabilistic distributions. Nondeterministic options are still present, because a rewrite step still requires selecting a rule and the position where it is applied. Statistical model-checking methods can be used to analyze quantitative properties of these models, but only when all sources of unquantified nondeterminism have been completely removed. In order to achieve this goal, its authors propose an actor-based framework consisting of object-oriented modules that satisfy some restrictions and where message delays are subjected to probabilistic distributions. Another tool, PSMaude [BÖ13], lets the user quantify the nondeterminism with strategies that fix probabilities for selecting rules, positions, and substitutions using a specific syntax for each of them. This tool is based on Full Maude and includes a PCTL [HJ94] model checker implemented in Maude itself.

Two alternative probabilistic extensions of Maude are presented in the next pages, which are respectively based on the external model-checking techniques of Section 4.4 and on the Maude strategy language. We think that they are more flexible and easy to use than the precedents we have just mentioned.

- In the first extension, probabilities are added outside Maude to the rewrite graph using different methods, from assuming that all successors of a state are equiprobable to specifying their probabilities with a Maude term that is evaluated on every transition. This yields discrete-time Markov chains or Markov decision processes that can be verified by probabilistic model-checking methods through the `umaudemc` utility (see Section 4.4) using the PRISM tool [KNP11] as a backend. It can be applied to both strategy-controlled and uncontrolled Maude specifications.

- In the second alternative, the strategy language is augmented with two probabilistic combinators that allow selecting the strategy to continue with and instantiate variables by sampling. Models specified using this approach are intended to be simulated or analyzed by statistical model checking, unlike the previous ones that are verified by probabilistic model-checking techniques.

The concepts we have introduced in Section 2.7 will be used extensively within this chapter. Namely, it will be useful to remember the definitions of discrete-time Markov chain, Markov decision process, and the probabilistic logic PCTL [HJ94]. We should also recall how probabilities are induced on the set of executions of a discrete-time Markov chain, and how they determine the probability of temporal properties to be satisfied.

This chapter describes recent work that has been developed during a research stay in the Chair of Foundations of Software Reliability and Theoretical Computer Science led by Javier Esparza at the Technische Universität München. The tools described in the following are less mature than those included in previous chapters, and we provide fewer details of aspects that are subject to change. They are applied to represent and analyze population protocols and chemical reaction networks in Chapter 9.

5.1 Turning Maude specifications into probabilistic models

In Section 4.4, we have connected Maude to some external model checkers to extend the repertory of temporal logics that can be used on its specifications. The fundamental idea is that the rewriting graph produced by Maude along with the possibility of evaluating atomic propositions on its states can be exported as a Kripke structure that other model-checking tools can digest. Similarly, we have extended `umaudemc` here to support probabilistic model checking on Maude modules using the PRISM [KNP11] tool as a backend. Instead of plain Kripke structures, we should generate probabilistic models, for which some additional information is required. Probabilistic model checking has been made available via a new `pcheck` command, whose arguments almost coincide with that of the classical `check`:

`umaudemc pcheck` ⟨ *file name* ⟩ ⟨ *initial term* ⟩ ⟨ *formula* ⟩ [⟨ *strategy* ⟩]
[`--assign` ⟨ *method* ⟩]

Models are prepared as for the classical model checkers, and probabilities are assigned by some alternative configurable methods that are described in the next paragraph. These methods turn the nondeterministic Maude model into a discrete-time Markov chain or a Markov decision process, where the probability that a temporal formula is satisfied can be calculated as explained in Section 2.7. Formulae can be expressed in both LTL and PCTL, and temporal operators can be annotated with TCTL-like bounds on the number of steps. Their syntax is an extension of the default `LTL` module with these new operators:

```
op _U__ : Formula Bound Formula -> Formula [ctor prec 63 format (d r b o d)] .
op _R__ : Formula Bound Formula -> Formula [ctor prec 63 format (d r b o d)] .

op <>__ : Bound Formula -> Formula [prec 53 format (r b o d)] .
op []__ : Bound Formula -> Formula [prec 53 format (r d b o d)] .
op _W__ : Bound Formula Formula -> Formula [prec 63 format (d r b o d)] .

op P__ : Bound Formula -> Formula [ctor prec 65 format (r b o d)] .

op [_,_] : Float Float -> Bound [ctor] .
op ≤_  : Float -> Bound [ctor] .
op ≤_  : Nat -> Bound [ctor] .
*** the same for <, ≥, >
```

Remember that $\mathbf{P}_I\ \varphi$ holds if the probability that φ is satisfied is in the interval $I \subseteq [0,1]$, and that step-bounded LTL operators are equivalent to expressions using $\bigcirc$ in the standard subset.

Assigning or distributing probabilities among the successors of each state is possible using several alternative methods enumerated below. Since they do not exploit any particular feature of rewriting logic, we will better describe them in the context of an abstract labeled transition system $(S, A, \to)$. Applying these methods yields either a discrete-time Markov chain (S, P, P_0) or a Markov decision process (S, A, P, P_0) abusing of the name P, where the states S and labels

A are those of the original model. Since properties are usually checked from a given initial state $s_0 \in S$, P_0 can be set as $P_0(s_0) = 1$ and $P_0(s) = 0$ for all $s \neq s_0$. Let us denote by $\mathrm{sc}(s) = |\{s' \in S : s \to s'\}|$ the number of successors of the state s, by $\mathrm{sc}(s, a) = |\{s' \in S : s \to^a s'\}|$ its labeled counterpart, and by $\mathrm{enabled}(s) = \{a \in A : \exists s' \in S \; s \to^a s'\}$ the set of actions enabled in s. The probability assignment methods are the following:

- `uniform` assigns uniform probabilities among the successors, as $P(s, s') = 1\,/\,\mathrm{sc}(s)$ if $s \to s'$ and $P(s, s') = 0$ otherwise. This is the default method when no `--assign` option is provided.

- `mdp-uniform` applies the same assignment as in the previous case, but leaving the choice of the action nondeterministic. Then, $P(s, a, s') = 1\,/\,\mathrm{sc}(s, a)$ if $s \to^a s'$ and $P(s, a, s') = 0$ otherwise. This yields an MDP instead of a DTMC, as its name suggests.

- `uaction(`a_1`=`w_1`, ..., `a_m`=`w_m`, `a_{m+1}`.p=`p_1`, ...)` distributes the probability among the successors of a state in two phases. First, the probability is shared among the enabled actions according to the arguments of `uaction`, and then the probability assigned to each action is divided up among its successors. More precisely, actions are partitioned into A_w and A_p depending on whether a probability or a weight has been assigned to them. These assignments are given by the functions $w : A_w \to \mathbb{R}^+$ and $p : A_p \to [0, 1]$. Let the assigned probability at s be $\mathrm{ap}(s) = \sum_{a \in \mathrm{enabled}(s) \cap A_p} p(a)$, and the total weight at s be $\mathrm{tw}(s) = \sum_{a \in \mathrm{enabled}(s) \cap A_w} w(a)$. A valid assignment must satisfy $\mathrm{ap}(s) \leq 1$, and $\mathrm{ap}(s) = 1$ if $\mathrm{tw}(s) = 0$. In other words, on a given state $s \in S$, all enabled actions subject to a probability are required to sum at most one, and enabled actions bound to weights share the remaining probability in proportion to their weights. Hence,

$$P(s, s') = \sum_{s \to^a s'}^{a \in A_p} \frac{p(a)}{\mathrm{sc}(s, a)} + \frac{1 - \mathrm{ap}(s)}{\mathrm{tw}(s)} \sum_{s \to^a s'}^{a \in A_w} \frac{w(a)}{\mathrm{sc}(s, a)},$$

  understanding that the second part is empty if $\mathrm{tw}(s) = 0$.

- Another family of methods can be described by a weight function $w : S \times S \to \mathbb{R}^+$ on transitions, assuming that for all $s \in S$ there is an $s' \in S$ such that $w(s, s') > 0$ and that $w(s, s') > 0$ implies $s \to s'$. Then, $P(s, s') = w(s, s')\,/\,\sum_{t \in S} w(s, t)$. Multiple methods are considered depending on how this function is specified:

  – `metadata` reads the weights as numeric literals written in the free-text metadata attribute of the Maude rule. In case multiple rules produce the same successor, the weight of one of them is chosen arbitrarily. We could admit expressions on the variables of the rule instead of literals for a greater generality, but this is not currently supported.

  – `mdp-metadata` does the same as the previous method but only among the successors for each rule label, leaving the choice of the action nondeterministic.

  – `term(`t`)` specifies w as a Maude expression that evaluates to a `Nat` or `Float` number. This expression may depend on the initial and final terms of the transition by means of the variables `L` and `R` respectively, and also on the label of the applied rule through a variable `A` of sort `Qid`.

These general methods can be applied to the uncontrolled rewriting graph $\mathcal{M}_t$ or to a strategy-controlled one $\mathcal{M}'_{\alpha,t}$. Like in Section 4.4, the states of the transition system may be merged so that probabilities are not interfered by the multiplication of successors by the strategy.

Proposition 5.1. *Given a labeled (or unlabeled when appropriate) transition system $(S, A, \to)$ and the additional data required by the methods,* `uniform`*,* `uaction`*,* `metadata`*, and* `term` *produce a discrete-time Markov chain (S, P, P_0). On the other hand, the methods* `mdp-uniform` *and* `mdp-metadata` *produce a Markov decision process (S, A, P, P_0).*

Notice that the probabilities obtained for a given Kripke structure $\mathcal{K}$ and a strategy λ in the unwinding $\mathcal{U}(\mathcal{K}, \lambda)$ do not necessarily coincide with the conditional probabilities in $\mathcal{K}$ under $E(\lambda)$ after applying an assignment method to both transition systems. For example, consider a Kripke structure with states $\{O, A, A', B\}$ such that $O \to^a A$, $O \to^a A'$, and $O \to^b B$. Under the `uaction` assignment method with equal weights for a and b, the executions OA^ω, OA'^ω, and OB^ω would respectively have probabilities 1/4, 1/4, and 1/2. If the system is restricted by a strategy $\{OA^\omega, OB^\omega\}$, the conditional probabilities of these executions should be 1/3 and 2/3. However, if we apply the same `uaction` method to the unwinding and consider the projections by last, OA^ω and OB^ω would have both probability 1/2.

Let us illustrate the `pcheck` command and its assignment methods with the following toy example modeling a coin toss:

```
mod COIN is
  sort Coin .
  ops head tail : -> Coin [ctor] .

  vars C C' : Coin .

  rl [thead] : C => head [metadata "8"] .
  rl [ttail] : C => tail [metadata "5"] .
endm
```

Tossing a coin with the `thead` and `ttail` rules leaves either its `head` or its `tail` upturned. Since the result is independent of the previous state of the coin, the executions of this model are the free words over the {`head`, `tail`} alphabet. First, we suppose the coin is balanced and both sides are equally likely to appear. This can be specified with the `uniform` assignment method in the `pcheck` command, which is the default when no `--assign` argument is passed. We ask for the probability of obtaining four head faces in a row by checking the LTL property $\square^{\leq 4} head$,[1] where the faces of the coin have been defined as atomic propositions that hold only when the state coincides with them. Notice that the exponent of the temporal operator refers to the discrete execution time, which is zero in the initial term and advances one by one in each step. Hence, $\square^{\leq 4} head$ says that *head* holds in the initial state (at time 0) and in four more states (at times 1, 2, 3, and 4), so that four heads have been obtained in addition to the initial one.

```
$ umaudemc pcheck coin head '[] <= 4 head'
Result: 0.0625
```

This is the expected result we could have easily calculated by hand. In effect, even for any number of tosses n, obtaining a head has probability one half and the n tosses are independent, so their probability is the product $(1/2)^n$. This implies that obtaining infinitely many heads in a row is almost impossible, since this number tends to zero as n grows. Its dual property $\diamond\, tail$ should then have probability 1:

```
$ umaudemc pcheck coin head '<> tail'
Result: 1.0
```

Another interesting feature of the `pcheck` command, directly provided by PRISM, is the `--steps` option that calculates the expected number of steps until a reachability or co-safe formulae is satisfied.

[1] We say that $\square^{\leq 4} head$ is an LTL formula even though $\square^{\leq 4}$ is not an LTL operator, because that formula is equivalent to $head \wedge \bigcirc\, head \wedge \cdots \wedge \bigcirc\bigcirc\bigcirc\bigcirc\, head$.

```
$ umaudemc pcheck coin head '<> tail' --steps
Result: 2.0
```

This can also be calculated algebraically, because the probability of seeing the first tail after n steps is 2^{-n} where $n-1$ of these halves come from the probability of obtaining $n-1$ heads and the last half is the probability of obtaining a tail. The expected value of the number of steps is then by definition $\sum_{k=0}^{\infty} k 2^{-k} = 2$.

We stop assuming now that the coin is balanced, and we will consider that one face is more likely to appear than the other. In the rules of the `COIN` module, we have attached some `metadata` attributes, the numbers 8 for `thead` and 5 for `ttail`. Using the `metadata` method, these numbers will be the weights or the likeliness of these rules to be applied. Since both rules are always enabled, these weights are translated to the probabilities 8/13 and 5/13, so the probability of obtaining 4 heads is $(8/13)^4$.

```
$ umaudemc pcheck coin head '[] ≤ 4 head' --assign metadata
Result: 0.1434123455061096
```

Similarly, the expected number of steps until obtaining a `tail` has grown too.

```
$ umaudemc pcheck coin head '<> tail' --assign metadata --steps
Result: 2.5999999999999996
```

An alternative method to specify the same weights is `uaction`, which receives them as arguments outside the Maude module. The same results must be obtained.

```
$ umaudemc pcheck coin head '[] ≤ 4 head' --assign 'uaction(thead=8, ttail=5)'
Result: 0.1434123455061096
$ umaudemc pcheck coin head '<> tail' --assign 'uaction(thead=8, ttail=5)'
                                      --steps
Result: 2.5999999999999996
```

Finally, we can assign arbitrary weights to the transitions with the `term` method. They may depend on the previous state and on the rule. For example, to express that obtaining the same face again is twice as likely, we can define the following function `inertia` in the `COIN` module and use it in the `term` assignment argument.

```
  op inertia : Coin Coin -> Nat .
  eq inertia(C, C') = if C == C' then 2 else 1 fi .
```

When calling this function as `term(inertia(L:Coin, R:Coin))` in the argument to `term`, the variables `L` and `R` will be replaced by the left- and righthand side of the transition respectively. Instead of defining `inertia`, we could have written the conditional expression directly on the command line.

```
$ umaudemc pcheck coin head '[] ≤ 4 head'
                  --assign 'term(inertia(L:Coin, R:Coin))'
Result: 0.19753086419753085
```

Under these conditions, obtaining four faces has a greater probability than in the balanced case too. Of course, every result we have obtained for this example can be proven algebraically without much effort, but more complex problems can be handled.

All the methods we have tested in the previous example quantify every nondeterministic choice, hence producing discrete-time Markov chains. In order to illustrate the verification of Markov decision processes, where nondeterministic and probabilistic actions are combined, let us introduce a more interesting yet unrealistic example of a postrider journey.

```
mod POSTHOUSE is
  protecting NAT .
```

```
   sorts Postman Posthouse Route .
   subsorts Postman Posthouse < Route .

   op postman : Nat Nat Nat -> Postman [ctor] .
   op house   : Nat Nat -> Posthouse [ctor] .
   op none    : -> Route [ctor] .
   op __      : Route Route -> Route [ctor assoc comm id: none] .

   vars E F N M : Nat .
   var  R       : Route .

   rl [advance] : postman(N, s(E), M) => postman(s(N), E, M) .
   rl [advance] : postman(N, s(E), M) => postman(N, 0, M) .

   crl [change] : postman(N, F, s(M)) house(N, E)
               => postman(N, E, M) house(N, E) if F < E .

   op initial : -> Route .
   eq initial = postman(0, 4, 3) house(2, 3) house(3, 5)
                house(7, 6) house(7, 2) house(11, 4) house(13, 5) .
 endm
```

The states of this model are routes consisting of several posthouses where a postman can stop to change horses in order to complete his duty on time. Posthouses are represented as terms `house(`n`, `e`)` where n is its mileage point in the route and e is the energy of its animals. The postman is another term `postman(`n`, `e`, `m`)` where n is again its position in the route, e is the energy of the horse he is currently riding, and m is the amount of money available to spent in the changes. Advancing a position in the route costs one unit of the horse energy with the first `advance` rule, and the postman will get blocked if his horse get exhausted. Moreover, the postman may also get blocked if an accident occurs due to the second `advance` rule. We will soon specify the probability of an accident, which will naturally depend on the freshness of the mount. In order to avoid them, the postman can change horses in a posthouse after paying a unit of his money with the `change` rule, whose condition avoids counterproductive exchanges.

The sources of nondeterminism of the `POSTHOUSE` module can be split in two parts: whether the `advance` or the `change` action is taken, and whether the first or the second rule of `advance` is applied. The first decision corresponds to the postman when planning the trip, while the possibility of an accident is a matter of chance. As a consequence, we want to keep the first choice nondeterministic while fixing probabilities for the second, thus obtaining a Markov decision process. In order to use the `mdp-term` method, we define a function that fixes the weights of the transitions depending on the energy attribute of the `postman`.

```
   op tiredness : Route Route -> Nat .
   eq tiredness(postman(N, E, M) L, postman(N, 0, M) R) =
        if E > 5 then 1 else sd(6, E) fi  .
   eq tiredness(L, R) = 200 [owise] .
```

Notice that these weights are applied to each action separately, so the only possible application of the `change` rule will be given a weight of 200 without effect. They will be actually used only for the `advance` rule when `E` is greater than 1. The probability of an accident is 0,05% when the energy of the animal is greater than five, but can grow up to 1,96% when the horse becomes tired. For a given route, like `initial`, we want to know whether and with which probability it is possible to reach the destination, assuming the postman takes the best decisions possible. The reachability of a certain point in the route can be specified as an atomic proposition as usual.

```
   op reach : Nat -> Prop [ctor] .
```

```
  eq postman(N, E, M) R ⊨ reach(N) = true .
  eq R ⊨ reach(N) = false [owise] .
```

If the end of the route in `initial` is the position 15, we should check $\Diamond\, \mathit{reach}(15)$ by selecting the `mdp-term` method with the `tiredness` function specified above.

```
$ umaudemc pcheck posthouse initial '<> reach(15)'
                    --assign 'mdp-term(tiredness(L, R))'
Result: 0.0 to 0.4221323452495324
```

Two probabilities are obtained, the minimum and maximum among all possible ways of solving the nondeterministic choices, where the maximum coincides with an optimal strategy of the postman. Apart from the semantics of the example, there is no impediment to use the `term` method instead of `mdp-term` when invoking `pcheck`, although this will not have any reasonable interpretation in this context.

Some more examples are available in the example collection. In Chapter 9, we discuss the specification and verification of population protocols and chemical reaction networks using this approach.

5.1.1 Implementation notes

As already mentioned, the implementation of the `pcheck` command relies on the external model checker PRISM [KNP11] to which it is connected following the procedures described in Section 4.4 for the other classical backends. Connecting an additional probabilistic backend like Storm [HJK+21] would be relatively straightforward. Given a Maude module, an initial term, optionally a strategy, and a probability specification with the `assign` argument, the `umaudemc` tool constructs a discrete-time Markov chain or a Markov decision process module in the PRISM language. For example, this is the model generated for the biased coin example defined by the `metadata` or `uaction` methods.

```
dtmc
module M
  x : [0..2] init 0;

  [] x=1 → 0.6153846153846154:(x'=0) + 0.38461538461538464:(x'=1);
  [] x=0 → 0.6153846153846154:(x'=0) + 0.38461538461538464:(x'=1);
endmodule

label "tail" = x=1;

rewards
  [] true: 1;
endrewards
```

In addition to the module, the atomic properties included in the input formula are evaluated and defined as labels, which are then used in the PRISM formula in which the Maude one is translated. Moreover, if the `--steps` flag is used, a reward specification that counts one for each state is inserted too. Another argument `--reward` is admitted by `pcheck` to indicate more general rewards as Maude functions to be evaluated on the states. The `maude` Python library is used to explore the strategy-free or strategy-aware Maude model and to evaluate the terms passed to the probability assignment methods. As a consequence, the PRISM backend should be placed in the rightmost part of Figure 4.7, among all other Python-based backends like NuSMV and Spot. Finally, the `prism` command is called with the appropriate input file and its output is parsed to be presented to the user as shown in the previous examples.

Like in Section 4.4, we want that `umaudemc` abstracts the details of the communication with the external backends, so the language of temporal formulae is specified and parsed in Maude as an extension of the `LTL` module with the operators that have appeared at the beginning of the section.

5.2 A probabilistic extension of the Maude strategy language

Extending the strategy language with probabilistic operators is another way of adding probabilities on top of an existing rule-based specification. As mentioned in Section 2.7, other strategy languages like ELAN and Porgy already count with a probabilistic choice operator very similar to the **`choice`** combinator that we are proposing in this section. Apart from this operator, we have extended the strategy language with another one that allows sampling variables subject to some probability distributions.

⟨*Strat*⟩ ::= `choice(` ⟨*TermStratList*⟩ `)`
  | `sample` ⟨*VarId*⟩ := ⟨*DistribId*⟩ `(` ⟨*TermList*⟩ `)` `in` ⟨*Strat*⟩

⟨*TermStratList*⟩ ::= ⟨*Term*⟩ `:` ⟨*Strat*⟩
  | ⟨*TermStratList*⟩ `,` ⟨*TermStratList*⟩

⟨*DistribId*⟩ ::= `bernoulli` | `uniform` | `normal` | `gamma` | `exp`

On the one hand, the **`choice`** combinator **`choice`**`(`w_1 `:` α_1`, …,` w_n `:` α_n`)` randomly chooses a strategy α_k in proportion to their weights w_k. When this operator is evaluated, the weight terms are instantiated $\theta(w_k)$ with the current variable assignment and reduced by the equations to literals w'_k of sort `Nat` or `Float`. The probability of executing the strategy α_k is then $w'_k \mathbin{/} \sum_{l=1}^{n} w'_l$.
On the other hand, **`sample`** `X :=` *distrib*`(`t_1`, …,` t_n`)` **`in`** α samples the variable `X` out of a distribution *distrib* with a given set of parameters t_1 to t_n. This variable can be later used in the nested strategy α. A hard-coded set of distributions is offered, including `bernoulli(`p`)` for the Bernoulli distribution of parameter p, `uniform(`a`,` b`)` for a uniform real distribution in $[a, b]$, `normal(`μ`,` σ`)` for a normal distribution with mean μ and standard deviation σ, `gamma(`α`,` β`)` for a gamma distribution, and `exp(`λ`)` for an exponential one.
While the first combinator is compatible with deriving a DTMC or an MDP as in the previous section, the second is clearly intended to stochastic simulations where the real-valued variables are sampled to determine the next step. As a consequence, when these combinators are executed, a single strategy in the **`choice`** operator or a single variable value in the **`sample`** strategy are randomly chosen to continue the execution of the strategy.
The two new operators are implemented in C++ in our modified version of the Maude interpreter, and so they can be used in the `srewrite` command and in the strategy-controlled `search` command, taking into account that the search will not be exhaustive but a single exploration branch will be opened at random when executing one of the new operators. Moreover, **`choice`** and **`sample`** have been defined at the metalevel in the `META-STRATEGY` module like the other strategies, so that they can be manipulated as data and executed by the `metaSrewrite` descent function.

```
sort ChoiceEntry < ChoiceMap .
subsort ChoiceEntry < ChoiceMap .

op choice : ChoiceMap -> Strategy [ctor] .
op sample_:=_in_ : Variable Term Strategy -> Strategy [ctor] .

op _:_ : Term Strategy -> ChoiceEntry [ctor prec 21] .
op _,_ : ChoiceMap ChoiceMap -> ChoiceMap [ctor assoc comm prec 61] .
```

Multiple executions of a probabilistic strategy on the same session of the Maude interpreter may produce different results, since the random source used to decide the **`choice`** and **`sample`** operators is not reset between executions of the `srewrite` commands. Nevertheless, the behavior in the whole session is deterministic and fixed by default, since it depends on the `random-seed` argument passed to Maude in the command line. Monte Carlo simulations can be performed by executing repeatedly the `srewrite` command in the same Maude session and gathering the results. This can be done programmatically using the `maude` Python library when compiled with the probabilistic extension. However, the soundness of the frequencies obtained during the simulation requires that some conditions are met:

- Strategies should not contain nondeterministic behavior, at least in the point where it can be observed. For example, if we only look at the final results of the strategy, confluent nondeterminism during the computation can be allowed as long as a single solution is obtained for each execution.

- Simulations should not be interrupted like standard strategy executions when repeated work is detected. In the nondeterministic case, an exhaustive search from an already visited state yields the same results every time, but in stochastic simulations different results can be obtained.

The second requirement is ensured by the implementation, but the first must be checked by the user. In order to execute simulations more efficiently, an auxiliary command `simaude` is provided.

`simaude` ⟨ *Maude file* ⟩ ⟨ *initial term* ⟩ ⟨ *strategy* ⟩ [`-n` ⟨ *simulations* ⟩] [`-j` ⟨ *jobs* ⟩]

It does what we have mentioned in the previous paragraph, evaluating a strategy multiple times while taking account of the results, but it leverages the internal hashing of terms in the strategic search to do it more efficiently. The output of the `simaude` command is deterministic given an initial seed for the pseudorandom number generator with the `random-seed` command-line option. The total number of simulations can be passed with the `-n` option, and the number of parallel processes to carry out the simulation can be provided with `-j`. Since each execution is independent of the rest, parallelizing the simulation is straightforward and can accelerate the experiment. However, the speed-up is not proportional to the number of jobs, since some time is spent in communicating the results to the parent process.

An example of the `sample` operator can be read in Section 5.2.1 and several applications of the **`choice`** operator are described in Chapter 9, but let us illustrate the latter operator with a simple example that has already appeared in the thesis, the river crossing puzzle in Section 4.3. Suppose that a player is trying to solve the puzzle doing unconscious moves at random, we want to know which is the probability of achieving this within a number of steps.

```
sd uniformStep := choice(1 : alone, 1 : wolf, 1 : goat, 1 : cabbage) .
```

Any action is equally desirable for this player with blinkers, so all of them have been assigned the same weight. However, some of these actions may fail in some states, since the `wolf`, `goat`, and `cabbage` rules require that the animal or plant being moved shares the same side of the river with the shepherd. The `uniformStep` can be refined to choose only among the rules that are available at a given moment.

```
smod RIVER-CHOICE is
  protecting RIVER-STRAT .

  sd uniformStep := matchrew R by R using choice(1 : alone,
                                      sameSide(R, wolf) : wolf,
                                      sameSide(R, goat) : goat,
                                   sameSide(R, cabbage) : cabbage) .

  var R : River .       var B : Being .       vars G G' : Group .

  op sameSide : River Being → Nat .
  eq sameSide(shepherd B G | G', B) = 1 .
  eq sameSide(R, B) = 0 [owise] .
```

In this new definition, the `choice` arguments are not numeric literals and they depend on the state matched by the variable `R` in the `matchrew`. These weights are produced by the `sameSide` function that takes the value 1 when the given being and the shepherd are in the same side of the river (when the rule is applicable), and 0 otherwise. In order to know the probability of solving the puzzle within a given number of steps, the `uniformStep` strategy should be repeated those many times.

```
  strat randomSearch : Nat @ River .
  sd randomSearch(0) := solved ? is_solved : not_solved .
  sd randomSearch(s N) := solved ? is_solved : ((eating or-else uniformStep) ;
                                                 randomSearch(N)) .

  op result : Bool → River .
  rl [is_solved]  : R ⇒ result(true) .
  rl [not_solved] : R ⇒ result(false) .
endsm
```

The `River` sort has been extended with two `result`(p) terms that summarize whether the solution has been found or not. The proper states of the game are transformed to these projections by the rules `is_solved` and `not_solved`. The `randomSearch` repeatedly executes the `eating` rule and the `uniformStep` strategy respecting their priorities, until the game is solved or the maximum number of steps is reached. We can execute a single simulation with the `srewrite` command in the Maude interpreter:

```
Maude> srew initial using randomSearch(100) .

Solution 1
rewrites: 403
result State: result(false)

No more solutions.
rewrites: 403
```

A single execution does not provide much information, but running the strategy many times allows estimating the probability of each result by the Monte Carlo method. This can be automatically done with the `simaude` command:

```
$ simaude river-choice.maude initial 'randomSearch(100)' -n 2000
Simulation finished after 2000 simulations and 785934 rewrites.

  result(false) 0.994
  result(true)  0.006
```

This estimation can be checked with the exact methods of the previous section, considering the LTL property $\Diamond$ *goal* under the `eagerEating` strategy.

```
$ umaudemc pcheck river initial '<> ≤ 100 goal' eagerEating
Result: 0.006211180124099635
```

There are little chances of solving the game like this, because a wrong move that leads the player into a risky state will make solving the game impossible, and wrong moves are possible in almost every step with great probability. However, if risky states are avoided by the player like in the `safe` strategy explained in Section 4.3, greater chances of solving the game are obtained.

```
$ umaudemc pcheck river initial '<> ≤ 100 goal' safe
Result: 0.8625619929029278
```

To check this property by simulation, a safe version of the `randomSearch` strategy is defined, where an auxiliary strategy `safeStep` tries `uniformStep` until a non-risky configuration is obtained.

```
strat randomSafeSearch : Nat @ River .
sd randomSafeSearch(0) := solved ? is_solved : not_solved .
sd randomSafeSearch(s N) := solved ? is_solved
                                   : (safeStep ; randomSafeSearch(N)) .

strat safeStep @ River .
sd safeStep := (uniformStep ; not(eating)) ? idle : safeStep .
```

Simulating this strategy confirms the probability obtained by the probabilistic model checker.

```
$ simaude river-choice.maude initial 'randomSafeSearch(100)' -n 1000
Simulation finished after 1000 simulations and 338110 rewrites.

  result(true)  0.853
  result(false) 0.147
```

With the `--steps` option of the `pcheck` command we can also find out that the expected number of steps until the game is solved at random is almost 55.

```
$ umaudemc pcheck river initial '<> goal' safe --steps
Result: 54.997596021137106 (relative error 9.778420421688145e-06)
```

Since this is a prerequisite for obtaining a finite expected value, it is almost sure that the goal will be eventually reached in an unbounded safe play. Hence, we can dare simulating the game without a step bound to estimate that expected value by statistical methods. For that purpose, the `River` sort is extended again as a tuple including a step counter, and a new strategy `randomCountSearch` is defined for simulation.

```
op <_,_> : River Nat -> River [ctor] .
op result : Nat -> State [ctor] .

crl [step]  : < R, N > => < R', N + 1 > if R => R' .

strat randomCountSearch @ River .
sd randomCountSearch := (matchrew < R, N > by R using solved) or-else
                           (step{safeStep} ; randomCountSearch) .
```

This strategy only stops when the solution is found. Steps are counted thanks to the `step` rule, whose rewriting condition is controlled by the former `safeStep` strategy that applies a single step to the first entry while the second entry is incremented by one.

```
$ simaude river-choice.maude '< initial, 0 >' 'randomCountSearch' -n 2000
Simulation finished after 2000 simulations and 962521 rewrites.

  < left | shepherd wolf goat cabbage right,85 >      0.005
  ...
  < left | shepherd wolf goat cabbage right,179 >     0.0005
  < left | shepherd wolf goat cabbage right,191 >     0.0005
```

The output of the command does not produce the expected value we want to calculate, but the complete lists of results of the strategy with their relative frequencies. However, this list can be easily parsed to calculate the weighted mean of the number of steps using an external program. The result is 55.84, which approximates the analytic result obtained before.

5.2.1 Expressing PMaude specifications with the probabilistic strategy language

PMaude [AMS06] is a probabilistic extension of Maude based on probabilistic rewrite theories, where rules may include some variables that are instantiated according to probability distributions that may depend on the matching substitution. More details have been given at the beginning of this chapter and in Section 2.7.

```
crl [pr] : t(x̄) ⇒ t′(x̄, ȳ) if C(x̄) with probability ȳ := π(x̄) .
```

With the extension of the strategy language just explained, it is possible to simulate these probabilistic rules. They can be defined in standard Maude as nonexecutable rules with $\bar{y}$ as free variables.

```
crl [pr] t(x̄) ⇒ t′(x̄, ȳ) if C(x̄) [nonexec] .
```

These variables can then be instantiated in the rule application operator with values obtained from some `sample` strategies featuring the distributions π of the original rule.

```
amatchrew S s.t. t(x̄) := S /\ C(x̄) by S
  using sample Y1 := π₁(x̄) in ⋯ sample Yn := πₙ(x̄) in
    pr[y₁ ← Y1, …, yₙ ← Yn] .
```

Since distributions may depend on the matching substitution, this needs to be recovered with the `amatchrew` that applies the rule with an initial substitution inside a nested chain of sample operators, one for each variable in $\bar{y}$. This encoding of the probabilistic rule is not specially efficient, since its lefthand side is matched and its condition evaluated redundantly, but in practice some parts could be simplified. Taking the disjunction of the previous construction for every rule in the model, we obtain a strategy that executes a step of a PMaude module.

We introduce here the first example of the PMaude paper [AMS06] to illustrate the `sample` operator and what has been explained in the previous paragraph. The module `EXPONENTIAL-CLOCK` specifies a clock holding a time count t and a battery charge c that can be either in a working `clock(`t`, `c`)` or a broken `broken(`t`, `c`)` state. The clock can be reset to zero with the `reset` rule or advanced with `advance`, which was a probabilistic rule in the original example with the annotations that have been retained as a comment. The period of the clock is given by an exponential distribution of parameter one, and a thousandth part of the battery is lost when it ticks. The charge determines the probability of the clock to become broken according to a Bernoulli distribution of parameter $c/1000$. Broken clocks cannot be further advanced or reset.

```
mod EXPONENTIAL-CLOCK is
  protecting FLOAT .

  sort Clock .
  op clock  : Float Float → Clock [ctor] .
  op broken : Float Float → Clock [ctor] .

  vars T C P BF : Float .
  var  B        : Bool .

  rl [advance] : clock(T, C) ⇒
       if B then clock(T + P, C - C / 1000.0)
            else broken(T, C - C / 1000.0) fi [nonexec] .
  *** with probability B := bernoulli(C / 1000.0) /\ P := exponential(1.0) .

  rl [reset] : clock(T, C) ⇒ clock(0.0, C) .
endm
```

As we have explained above, the strategy to apply the `advance` rule according to the aforementioned probabilities is the following `p-advance`, and `p-all` executes a single rewrite with any of the rules in the module.

```
strats p-advance p-all @ Clock .

sd p-advance := matchrew Clk s.t. clock(T, C) := Clk by Clk using (
                  sample BF := bernoulli(C / 1000.0) in
                  sample T  := exp(1.0) in advance[B <- BF == 1.0, T <- T] .

sd p-all := reset | p-advance .
```

Notice that the `bernoulli` distribution returns either `0.0` for `false` or `1.0` for `true`, and so the `B` variable of `advance` is instantiated with `BF == 1.0`.

```
Maude> srew clock(0.0, 5000.0) using p-all .

Solution 1
rewrites: 1
result Clock: clock(0.0, 5.0e+3)

Solution 2
rewrites: 8
result Clock: clock(2.9312150206607814, 4.995e+3)

No more solutions.
rewrites: 8
```

As we can see in the last execution, this module contains unquantified nondeterminism in the choice between the two available rules, so it cannot be analyzed quantitatively by simulation. However, the qualitative results of a simulation can still be interesting, and for instance, we can check whether a broken clock configuration is reachable with the strategy-controlled variant of the `search` command.

```
Maude> search [1] clock(0.0, 5000.0) =>+ broken(T, C) using p-all * .

Solution 1 (state 1298174)
states: 1298175  rewrites: 10374120
T --> 1.3406394527068862e+3
C --> 9.9663704215002917e+2
```

This search could have not terminated if the sample of the Bernoulli distribution always returned a positive answer. Nevertheless, this event has probability zero, since the reduction of the clock battery at each step makes the parameter of the Bernoulli distribution decrease, increasing the probability of breakage.

5.2.2 Implementation notes

As part of our version of the Maude interpreter extended with the strategy-aware model checker, the probabilistic **`choice`** and **`sample`** combinators are implemented in C++ like all other strategy combinators, and they have been similarly represented at the metalevel. When the **`choice`** combinator is executed, its weights are instantiated and reduced equationally. One strategy is then selected at random respecting the probabilities fixed by their weights. For the **`sample`** operator, the standard C++ library `random` is used to sample the given distribution, whose parameters are also instantiated and reduced equationally. The variable context of the nested expression is augmented with the sampled variable.

The `simaude` command uses a modified implementation of the strategic search class that can be reused for multiple executions while maintaining the same hash table of terms. This tool is written in C++ and directly linked with the implementation of Maude. Parallel simulations are executed in multiple processes that communicate their results to the main one via Unix sockets.

Since each process maintains its own hashing table for the strategic search, terms have to be passed in printed form between them.

Even though these new operators are implemented in C++ for efficiency, they can also be defined inside the standard strategy language using the special `random` operator provided by the `RANDOM` module of the Maude prelude.

```
op random : Nat -> Nat [special (...)] .
```

The term `random(`n`)` is reduced to the n-th element of a fixed sequence of pseudorandom numbers in the range $[0, 2^{32} - 1]$, whose initial seed is given by the `random-seed` argument of the Maude interpreter. Assuming all weights w_k are `Float` terms for simplicity, the strategy call `my-choice(RE,` $w_1 \cdots w_n$`)` with the following definitions implements an instance of the **`choice`**`(`w_1 `:` α_1`, ...,` w_n `:` α_n`)` operator:

```
vars RE N : Nat .       vars F W : Float .      var WL : FloatList .

strat my-choice : Nat FloatList @ State .
sd my-choice(RE, WL) := option(s RE,
        choice(0, (sum(WL) * float(random(RE))) / (2.0 ^ 32.0 - 1.0), WL)) .

op choice : Nat Float FloatList -> Nat .
eq choice(N, F, W) = N .
eq choice(N, F, W WL) = if F < W then N
                                else choice(s N, F - W, WL) fi [owise] .

op sum : FloatList -> Float .
eq sum(nil) = 0.0 .
eq sum(W WL) = W + sum(WL) .

strat option : Nat Nat @ State .
sd option(RE, 1) := α1 .
sd option(RE, n) := αn .
```

The selection of the strategy to be applied takes place in the `choice` function, which is fed by the random floating-point number in the range $[0, \sum_{k=1}^{n} w'_k]$ calculated in the `my-choice` strategy. This expression `(sum(WL) * float(random(RE))) / (2.0 ^ 32.0 - 1)` yields a random number uniformly distributed in that range, since the `sum` result is multiplied by the quotient of the random integer of 32 bits `random(RE)` and its maximum value. The sequence of floating-point numbers in `WL` partitions the sample interval and the argument `F` of `choice` must fall in one of these parts, hence choosing at random one of the strategies. As usual, the weights w_k in the call to `my-choice` may contain variables, which are instantiated from the context. The C++ implementation of the **`choice`** operator does not differ much from this.

Similarly, **`sample`** x **`in`** α can be implemented in the standard language by replacing the occurrences of x in α by expressions containing the `random` operator called with successively increasing enumerators. In the previous paragraph, we have seen how uniform distributions in $[0, B]$ can be obtained for any real number B, while translating this interval or limiting the sample to integers are easy variations. Any probabilistic distribution can be obtained as well with this procedure, which is followed by PMAUDE, but the Maude code to support them would be more complex and probably less efficient than the builtin `sample` primitive.

The fact that these operators can be expressed using the official strategy language poses the question whether a dedicated implementation is convenient. Since estimating quantitative properties of a model requires executing many simulations, having efficient implementations of these operators is interesting to speed them up. Moreover, the custom implementation with the **`choice`** operator requires carrying a pseudorandom number enumerator that complicates the definition of the strategies. At the opposite end of the scale, the custom implementation of the operators provide a greater flexibility, for example, to increase the repertory of probabilistic distributions that are supported.

Part III

Examples

Chapter 6

Examples

Using the strategy language and the model checkers introduced in Part II, we can write Maude specifications where the application of rules is controlled at a higher level using strategies and verify them against temporal properties. As part of this thesis, we have developed several strategy-based specifications and checked temporal properties on many of them, which have allowed evaluating the expressivity of the language and the performance of the model checkers. A selection of these examples is described in Part III and in particular in this chapter. Moreover, other authors have developed specifications and programs using the Maude strategy language. Some interesting examples were already written and published before this thesis and with the first Maude-based prototype. We have updated them to the current version of the language, fixing some bugs and making some improvements. In this chapter, several examples are chosen to cover the different possibilities of the tools and the various application areas where strategies have been found useful. They are the following:

1. Semantics of programming languages and other formalisms are addressed in the first section. We start with the simple and classical example of the λ-calculus, where multiple strategies have been traditionally considered to reduce terms. Afterward, a parameterized semantics is described for the functional programming language REC, where call by value and call by name are different strategies over the same base specification. Several and larger examples have been published in this field before this thesis like a semantics for the parallel functional Haskell-based language Eden [HVO07], two semantics for Cardelli and Gordon's ambient calculus [RSV06], and an implementation of Milner's calculus of communicating systems (CCS) [MPV07], where temporal properties can now be checked using our tools. More recently, the current version of the strategy language and its LTL model checker have been used to verify properties on smart contracts written in the language BitML for the Bitcoin blockchain [ABL+19]. We have also written an interpreter for a logical programming language similar to Prolog, where strategies conduct the search and implement negation and cut [CDE+20].

   Additionally, this chapter demonstrates the support for parameterized strategy specifications in Maude. More examples of this kind can be found in Section 6.5 and in [RMP+19c], including an implementation of the simplex algorithm for linear programming with alternative cycle prevention strategies that can be tested with the model checker, and word wrapping and fractal generation algorithms.

2. Section 6.2 is a simple example to illustrate the usage of the strategy-aware model checker and how a system controlled by different strategies satisfies different properties. The classical concurrency problem of the dining philosophers is used for this purpose. Since this specification can be easily extended to any desired size, it is used in Chapter 10 to evaluate the scalability of the model checker.

3. A more realistic example dealing with concurrency is in Section 6.3, where strategies are identified with the scheduling policies of an operating system in which multiple processes share a single processor. Some properties of the concurrent programs running on it are satisfied regardless of the scheduling policy, but some require additional restrictions to hold. The model checkers for branching-time properties in Section 4.4 and the prototype respecting the Kleene star semantics in Section 4.5 are used too.

4. Communication protocols are addressed in Section 6.4 with the *Routing Information Protocol* (RIP). Strategies are used in an object-oriented specification to control what happens during an iteration of the routing algorithm, to broadcast messages to all their recipients, and to purposely introduce failures for testing the protocol. Various temporal properties are checked on different networks and situations, and the support for opaque strategies of the model checker in Section 4.3.1 is applied.

5. Controlling deduction procedures based on a given set of rules is one of the paradigmatic use cases for strategies, following the principle of separation of concerns enunciated in the motto *Algorithm* = *Logic* + *Control* [Kow79]. The Knuth-Bendix and other equational completion procedures in Section 6.5 are a complete redesign of a previous specification [VM11] that used the Maude-based prototype of the strategy language, which was based in turn on previous work by Lescanne [Les90]. The novelty of our specification is that it fixes a set of Maude rules for the four alternative completion procedures, and only the recursive strategies that apply them vary from one to another. All procedures yield correct completions by the soundness of the inference system, but only those that apply the rules efficiently can handle the most complex problems. A similar approach has been applied to the Martelli-Montanari unification algorithm available in the example collection, and in a previous example dealing with abstract congruence closure [EMM$^+$07]. More recently, the strategy language has also been used to specify different strategies for the DPLL SAT-solving procedure [Her20].

6. Strategies are pervasive in games. In Section 6.6, we introduce the 15 puzzle and implement a strategy to solve it as described in [Luc92]. The tic-tac-toe game is discussed in Section 7.4 by checking multiple combinations of player strategies. A Sudoku solver [SP07] was also programmed using the first prototype of the strategy language. While adapting this solver to the current version of the language, we have improved its performance and fixed some bugs.

Chapter 7 gathers other examples that take advantage of reflection and extend the strategy language with new features, and Chapter 8 presents a simulator and model checker for membrane systems using strategies. This simulator is inspired on a previous work [AL09] making a heterodox use of an earlier version of the strategy language, which has also been used to implement neural networks [SPV09]. Chapter 9 represents population protocols and chemical reaction networks in Maude using the probabilistic extension in Chapter 5.

All examples in this thesis, the old examples adapted to the current version of the language, and additional smaller examples that are not covered here can be found in the strategy language website [EMM$^+$21] and in the example collection on `github.com/fadoss/strat-examples`.

6.1 Strategies in the λ-calculus and functional languages

The λ-calculus [Bar14] is a formal system introduced by Alonzo Church in the 1930s to study the notion of computable function, which can be easily expressed and executed in Maude [MM96, MM02]. Its syntax is defined with two operators for the λ-abstraction `\_._` and the application `__`, and the only rule is `beta` for β-reduction. The auxiliary `subst` function is used to substitute a variable by a term.

$$\begin{tikzcd}
 & (\boldsymbol{K}\boldsymbol{I})\,\Omega \arrow[loop right] & \\
(\boldsymbol{K}\boldsymbol{I})\,\Omega \arrow[ur] \arrow[dr] & & \\
 & (\lambda y.\boldsymbol{I})\,\Omega \arrow[r] & \boldsymbol{I}
\end{tikzcd}$$

$$\begin{aligned}
\boldsymbol{K} &= \lambda x.\,(\lambda y.x) \\
\boldsymbol{I} &= \lambda x.\,x \\
\Omega &= (\lambda x.\,xx)(\lambda x.\,xx)
\end{aligned}$$

Figure 6.1: Two reduction paths from the λ-term $(\boldsymbol{K}\boldsymbol{I})\,\Omega$.

```
mod LAMBDA is
  sorts Var LambdaTerm .
  subsort Var < LambdaTerm .

  op \_._ : Var LambdaTerm -> LambdaTerm [ctor prec 40 gather (& E) ...] .
  op __   : LambdaTerm LambdaTerm -> LambdaTerm [ctor gather (E e)] .

  op subst : LambdaTerm Var LambdaTerm -> LambdaTerm .

  vars x y : Var .          vars M N O : LambdaTerm .

  eq subst(x, y, M) = if x == y then M else x fi .
  eq subst(M N, x, O) = subst(M, x, O) subst(N, x, O) .
  eq subst(\ x . M, y, N) = \ x . (if x =/= y then subst(M, y, N) else M fi) .

  rl [beta] : (\ x . M) N => subst(M, x, N) .
endm
```

Terms can be executed with the `rewrite` command and reduction paths can be explored with `search`. However, which β-redex is reduced first is an important choice. Some reductions may not terminate while others reach an irreducible term, as shown in Figure 6.1, even though this is unique by the Church-Rosser property. The normalization theorem says that "reducing the leftmost outermost redex first" always leads to a normal form in case it exists. This phrase is one of the reduction strategies that are traditionally considered for the λ-calculus and functional programming languages. We will write some of them in a strategy module controlling `LAMBDA`:

```
smod LAMBDA-STRATS is
  protecting LAMBDA .
  strats applicative normal byname byvalue @ LambdaTerm .
  vars x y z t : Var .        vars M N : LambdaTerm .
```

- Applicative order (inner rightmost redex first)

```
  sd applicative := matchrew \ x . M by M using applicative
                  | matchrew M N by N using applicative
                    or-else matchrew M N by M using applicative
                    or-else top(beta) .
```

- Normalizing strategy (outer leftmost redex first)

```
  sd normal := matchrew \ x . M by M using normal
             | top(beta) or-else matchrew M N by M using normal
                         or-else matchrew M N by N using normal .
```

- By name (normalizing, but no reduction inside abstraction)

```
  sd byname := top(beta) or-else matchrew M N by M using byname
                         or-else matchrew M N by N using byname .
```

- By value (only outermost redexes and when argument is a value)

```
sd byvalue := (match (\ x . M) z | match (\ x . M) (\ y . N)) ; top(beta)
            | (matchrew M N by N using byname)
            | matchrew M N by M using byname .
```

These steps describing a single β-reduction should be repeated to exhaustively reduce the term. The normalization operator `!` will do it until a normal form is reached for the strategy. Notice that the normal forms for `byname` and `byvalue` are not real normal forms of the uncontrolled λ-calculus, since they refuse to reduce in certain positions. Applying these strategies to the term `(K I) Omega`, we see that a normal form is reached for all strategies but `applicative`.

```
Maude> srew (K I) Omega using normal ! .

Solution 1
rewrites: 17
result LambdaTerm: \ x . x

No more solutions.
rewrites: 17

Maude> srew (K I) Omega using applicative ! .

No solution.
rewrites: 11
```

The `applicative` strategy gets lost in the infinite reduction depicted in Figure 6.1. However, the `srewrite` command terminates without solution due to the cycle detection feature of the command. From `(K z) t` we get `z` using all strategies but `byvalue`, whose canonical form is the term unchanged.

6.1.1 The REC language

Strategies are also relevant for other functional programming languages, and namely the last two of the previous list, `byname` and `byvalue`. We will briefly illustrate this as well as parameterized strategy modules with the simple *Recursion equations* (REC) language [Win93; Chapter 9]. A REC program consists of a closed integer expression to evaluate and a set of integer function definitions of the form $f(x_1, \ldots, x_n) = \langle expr \rangle$. Expressions contain integer constants, variables, sums, products, subtractions, conditionals, and calls to any defined function. Hence, functions can be recursive and mutually recursive. An organized set of modules (see Figure 6.2) specifies the representation of REC programs, its basic rules, and the strategies. We will describe the most relevant parts omitting the details, since the full Maude specification is available in the collection. Integer expressions containing only literals are reduced equationally, since the sort `RecExpr` of expressions is defined as an extension of the builtin sort `Int`. Two rules are in charge of conditionals and function calls.

```
rl [apply] : Q(Args) => apply(find(Q, Defs), Args) [nonexec] .
crl [cond] : if C then E else F => if C == 0 then E else F fi if C : Int .
```

The rule `apply` replaces a function call `Q(Args)` by its definition according to the list of definitions `Defs`, substituting its variables by the call arguments using the equational function `apply`. This rule is not directly executable since `Defs`, absent in the lefthand side, must be provided from the context in the application strategy substitution, as we will see. Conditionals are resolved by the `cond` rule when their conditions have been reduced to an integer.

REC programs are executed following a `reduce` strategy that receives a list of function definitions as an argument. It is parameterized by a strategy `st` that is intended to expand function calls, and which we will later instantiate with `byvalue` and `byname` alternatives.

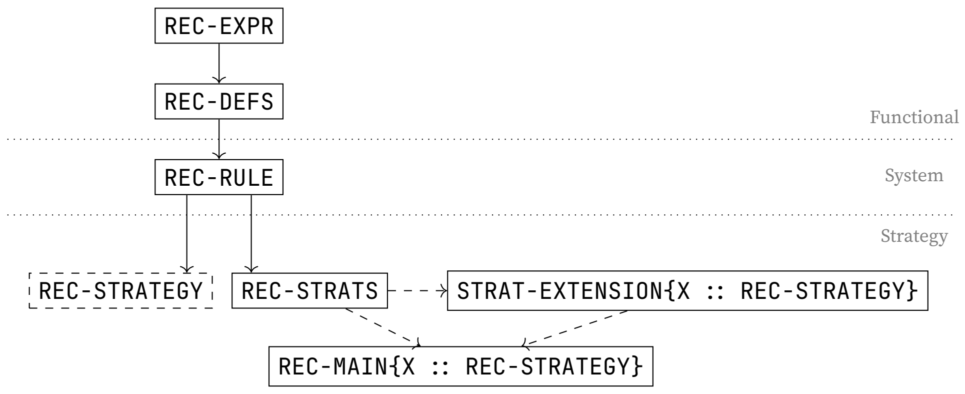


Figure 6.2: Module structure for the REC language specification.

```
sth REC-STRATEGY is
  including REC-RULE .
  strat st : List{FunctionDef} @ RecExpr .
endsth

smod REC-MAIN{X :: REC-STRATEGY} is
  strat reduce : List{FunctionDef} @ RecExpr .
  var FL : List{FunctionDef} .
  sd reduce(FL) := (cond or-else st(FL)) ! .
endsm
```

According to the `reduce` definition, conditionals and calls are reduced as long as possible, but conditionals are reduced first. This precedence is convenient, since function calls may appear anywhere in the expression, and a simple recursive example like the factorial

```
eq factorial = 'f('n) := if 'n then 1 else 'n * 'f('n-1) .
```

shows that expanding calls carelessly is problematic. In effect, the precedence of `cond` avoids the nonterminating reduction

$$\begin{aligned} \texttt{'f(0)} &\to_{st} \texttt{if 0 then 1 else 0 * 'f(-1)} \\ &\to_{st} \texttt{if 0 then 1 else 0 * (if -1 then 0 else 'f(-2))} \ \ldots \end{aligned}$$

This is still not enough when evaluating terms like `'f('f(0))`, and we must ensure in general not to reduce the branches of the conditional until its condition is solved, no matter if calling by value or by name. These two strategies only change how functions are called, so we do not want to repeat twice the strategies for traversing the expression structure. Given a strategy `st` that only handles function calls, we can derive a strategy `xst` able to reduce any term using a parameterized strategy module.[1]

```
smod STRAT-EXTENSION{X :: REC-STRATEGY} is
  strats xst xst-args : List{FunctionDef} @ RecExpr .

  vars E F G : RecExpr .       var FL : List{FunctionDef} .

  sd xst(FL) := (match Q(Args) ; (st(FL)
                 or-else matchrew Q(Args) by Args using xst-args(FL)))
  | matchrew E + F by E using xst(FL) or-else matchrew E + F by F using xst(FL)
```

[1]The previous version of this example published in [RMP+19c] was wrong because the extension was not applied inside the function arguments.

```
   | matchrew E * F by E using xst(FL) or-else matchrew E * F by F using xst(FL)
   | matchrew E - F by E using xst(FL) or-else matchrew E - F by F using xst(FL)
   | matchrew if E then F else G by E using xst(FL) .

  sd xst-args(FL) := matchrew E, Args by E using xst(FL)
                     or-else matchrew E, Args by Args using xst-args(FL) .
endsm
```

An auxiliary strategy `xst-args` applies the first possible reduction on the call arguments from left to right, and it is executed when the parameter strategy `st` fails. The module parameter `X` must adhere to the strategy theory `REC-STRATEGY`, like in `REC-MAIN`. By instantiating this skeleton, the alternative strategies `byname` and `byvalue` can now be defined succinctly in the strategy module `REC-STRATS`.

```
sd byname(FL)  := top(apply[Defs <- FL]) .
sd byvalue(FL) := match Q(LArgs) ; top(apply[Defs <- FL]) .
```

While both strategies execute `apply` on top, `byvalue` checks first whether all arguments of the call are literals, using the variable `LArgs` of sort `LiteralArguments` that only matches lists of integers. The reduction of the arguments relies on the extension strategy when it fails. Finally, we only have to instantiate the `REC-MAIN` module with the extensions of these strategies, obtained in turn by instantiating `STRAT-EXTENSION`. This can be achieved with two views `ByName` and `ByValue` from `REC-STRATEGY` to `REC-STRATS`, and a parameterized view from `REC-STRATEGY` to `STRAT-EXTENSION`.

```
view Extend{X :: REC-STRATEGY} from REC-STRATEGY to STRAT-EXTENSION{X} is
  strat st to xst .
endv

view ByName from REC-STRATEGY to REC-STRATS is
  strat st to byname .
endv
```

The module `REC-MAIN{Extend{ByName}}` defines a strategy `reduce` ready to evaluate REC programs with call by name according to the definitions received as arguments, and the same can be done for call by value. For example, the Ackermann function is defined by the following equation

```
eq ackermann = 'A('m, 'n) := if 'm then 'n + 1 else
  (if 'n then 'A('m - 1, 1) else 'A('m - 1, 'A('m, 'n - 1))) .
```

And then the expression $1 + A(2, 3)$ can be reduced to obtain 10 with both semantics.

```
Maude> srew 1 + 'A(2, 3) using reduce(ackermann) .

Solution 1
rewrites: 147450
result NzNat: 10

No more solutions.
rewrites: 147450
```

Only the number of rewrites changes in this case, since call by value is faster and only takes 7110 rewrites. The reason is that the arguments that are copied in the Ackermann definition are evaluated only once. More substantial differences between call by value and call by name are appreciated with the following definition,

```
eq nameonly = ('f('x) := 'f('x) + 1) ('g('x) := 7) .
```

since $g(f(0))$ will be evaluated to 7 by name, and it will not terminate by value.

REC programs can also be model checked. Following the typical procedure explained in Section 4.3, we have defined a module `REC-PREDS` extending `REC-RULE` with some atomic propositions, and then merged this with the instances of `REC-MAIN` for the different semantics. Without going into greater detail, two atomic propositions are defined: `isLiteral` that holds whenever the state is a literal, and `literalCond` that is satisfied when there is a conditional with a literal condition in the expression. The LTL property $\Diamond$ isLiteral is satisfied in all examples mentioned in this section and for all strategies, except for `'g('f(0))` and `nameonly` by value, where the model checker does not terminate. The graph of the program evaluation can be seen with the `umaudemc graph` command with an optional depth limit. Moreover, if the initial integer expression contains a variable, this property does not hold. A more interesting property is the μ-calculus formula $\nu Z.(\mathit{literalCond} \to ([\texttt{apply}]\,\bot \wedge \langle\texttt{cond}\rangle\top)) \wedge [\cdot]\,Z$, meaning that along every execution, in any state where there is a literal condition, `cond` can always and `apply` can never be applied. This property is guaranteed by the definition of `reduce`, so it holds for every instance and for every example.

```
$ umaudemc check recfns-mc.maude "1 + 'A(2, 3)" \
  'nu Z . (literalCond → ([ apply ] False /\ < cond > True)) /\ [.] Z' \
  'redName(ackermann)'
The property is satisfied in the initial state
(4790 system states, 168614 rewrites, 43117 game states)

$ umaudemc check recfns-mc.maude ... 'redValue(ackermann)'
The property is satisfied in the initial state
(157 system states, 7715 rewrites, 1420 game states)
```

6.2 The philosophers problem

The dining philosophers problem [Hoa85] is a classical concurrency problem, originally proposed by C.A.R. Hoare based on an exam exercise by E. Dijkstra. Five numbered philosophers are sat at a circular table around an endless bowl of spaghetti, and a golden fork is laid between each two contiguous philosophers. Although their main task is thinking, they should eat sometime to avoid getting starved, for what they need the two forks at both their sides. They should take one at a time and then put down when they have finished. The problem is that there are only five forks for five philosophers.

In the Maude specification, a philosopher is represented as a triple describing both hands contents (a fork ψ or nothing o) and an identifier. They can take the available forks at their sides with the `left` and `right` rules, and put them back with `release`. Since a circular table is represented by a list, we adopt the convention that the fork between the last and first philosophers is on the right, which is ensured by an equation and supported by a second `left` rule that allows the first philosopher to take this fork.

```
mod PHILOSOPHERS-DINNER is
  protecting NAT .

  sorts Obj Phil Being List Table .
  subsorts Obj Phil < Being < List .

  ops o ψ : -> Obj [ctor] .
  op (_|_|_) : Obj Nat Obj -> Phil [ctor] .
  op empty : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: empty] .
  op <_> : List -> Table [ctor] .
  var Id : Nat .
  var P  : Phil .
  var X  : Obj .
  var L  : List .

  rl [left]  : ψ (o | Id | X)       =>   (ψ | Id | X) .
  rl [right] :   (X | Id | o) ψ     =>   (X | Id | ψ) .
  rl [left]  : < (o | Id | X) L ψ > => < (ψ | Id | X) L > .
  rl [release] : (ψ | Id | ψ) => ψ (o | Id | o) ψ .

  eq < ψ L P > = < L P ψ > .
endm
```

Following the procedure described in Figure 4.4, this system module is the complete specification `M` of the uncontrolled model. The extension `DINNER-PREDS` below specifies the atomic propositions that will be used to describe properties of the problem behavior: a parameterized collection `eats(`n`)` meaning "the philosopher n eats", and `used(`n`)` standing for "the fork at the right of philosopher n is being used".

```
mod DINNER-PREDS is
  protecting PHILOSOPHERS-DINNER .
  including SATISFACTION .

  subsort Table < State .
  ops eats used : Nat -> Prop [ctor] .

  var  Id  : Nat .
  var  X   : Obj .
  vars L R : List .

  eq < L (ψ | Id | ψ) R > |= eats(Id) = true .
  eq < L > |= eats(Id) = false [owise] .

  eq < L (X | Id | o) ψ R > |= used(Id) = false .
  eq < L > |= used(Id) = true [owise] .
endm
```

Notice that `Table` is declared as a subsort of `State`, and equations are used to define the satisfaction of the atomic propositions on every state.

The uncontrolled execution of this system is not satisfactory for the philosophers integrity, as we will see soon, so some restrictions are specified using strategies. These are defined in a strategy module `DINNER-STRAT` that extends and controls `PHILOSOPHERS-DINNER`.

```
smod DINNER-STRAT is
  protecting PHILOSOPHERS-DINNER .

  strats free parity turns @ Table .
  strat turns : Nat Nat @ Table .

  var  T    : Table .
  vars L L' : List .
  vars K Id : Nat .
  var  N    : NzNat .
```

The first strategy, `free`, is the recursive and exhaustive application of all the rules in the module, and so it behaves like the built-in strategy of the `rewrite` command.

```
sd free := all ? free : idle .
```

Assuming that the philosophers in the table are numbered consecutively from zero, the equivalent of the solution proposed by Dijkstra to solve the original exam exercise is the `parity` strategy. It forces the diners to take first the fork at a fixed side, which is alternative for evens and odds, i.e. for neighbors. This restriction groups the philosophers in pairs where they compete for the middle fork, and only the one with this fork will try to obtain the outer fork shared with another couple, hence not impeding their other neighbors to take both forks and eat.

```
sd parity := (release
  *** The even take the left fork first
  | (amatchrew L s.t. ψ (o | Id | o) := L /\ 2 divides Id
       by L using left)
  | left[Id <- 0]
  *** The odd take the right fork first
  | (amatchrew L s.t. (o | Id | o) ψ := L /\ not (2 divides Id)
       by L using right)
  *** When they already have one, they take the other fork
  | (amatchrew L s.t. (ψ | Id | o) ψ := L by L using right)
  | (matchrew T s.t. < L (o | Id | ψ) L' > := T
       by T using left[Id <- Id])
  ) ? parity : idle .
```

The last strategy, `turns`, iterates through the philosophers in a loop, making them eat in turns. The strategy can be improved by allowing more than one philosopher to eat in parallel (with five philosophers, two can eat at each turn).

```
  sd turns(K, N) := left[Id <- K] ; right[Id <- K] ; release ;
                    turns(s(K) rem N, N) .
  sd turns := matchrew T s.t. < L (X | Id | Y) ψ > := T
               by T using turns(0, s(Id)) .
endsm
```

The argument `N` of the first `turns` strategy is the number of philosophers at the table, and `K` is the cyclic index to the current one. Their initial values are filled by the overload without arguments, which obtains the number of philosophers from the initial term.

Finally, the strategy specification in `DINNER-STRAT` is merged with the property specification in `DINNER-PREDS`. In the same module, an `initial` operator is defined to build the initial table with the given number of philosophers, which is five by default `< (o | 0 | o) ψ ··· (o | 4 | o) ψ >`. The rules and strategies of the model are valid regardless of the number of philosophers, which is determined by the initial term.

```
smod DINNER-SCHECK is
  protecting DINNER-STRAT .
  protecting DINNER-PREDS .

  op initial     :     -> Table .
  op initial     : Nat -> Table .
  op initialList : Nat -> List .

  eq initial = initial(5) .
  eq initial(N) = < initialList(N) > .
  eq initialList(0) = empty .
  eq initialList(s(N)) = initialList(N) (o | N | o) ψ .
```

```
endsm
```

Now, we can start model checking. The property that would guarantee the survival of the philosophers is the LTL property $\Box \bigwedge_{k=0}^{4} \Diamond\, \texttt{eats}(k)$, but the unrestricted system does not even satisfy the weaker non-deadlock requirement $\Box\Diamond \bigvee_{k=0}^{4} \texttt{eats}(k)$.

```
Maude> red modelCheck(initial, <> (eats(0) \/ … \/ eats(4)), 'free) .
rewrites: 120
result ModelCheckResult: counterexample(
  {< (o |0| o) ψ (o |1| o) ψ (o |2| o) ψ (o |3| o) ψ (o |4| o) ψ >,'left}
  {< (ψ |0| o) ψ (o |1| o) ψ (o |2| o) ψ (o |3| o) ψ (o |4| o) >,'left}
  {< (ψ |0| o) (ψ |1| o) ψ (o |2| o) ψ (o |3| o) ψ (o |4| o) >,'left}
  {< (ψ |0| o) (ψ |1| o) (ψ |2| o) ψ (o |3| o) ψ (o |4| o) >,'left}
  {< (ψ |0| o) (ψ |1| o) (ψ |2| o) (ψ |3| o) ψ (o |4| o) >,'left},
  {< (ψ |0| o) (ψ |1| o) (ψ |2| o) (ψ |3| o) (ψ |4| o) >,solution})
```

In this counterexample, every philosopher takes the left fork before anyone can take the right one and eat. Using the strategy-controlled model checker with the `free` strategy, as we have done, is equivalent to using the standard model checker without strategy. Indeed, the same counterexample is obtained, although with `deadlock` instead of `solution` as the last transition label. Deadlocks are avoided with the `parity` strategy:

```
Maude> red modelCheck(initial, [] <> (eats(0) \/ … \/ eats(4)), 'parity) .
rewrites: 1005
result Bool: true
```

However, `parity` does not guarantee that no philosopher starves.

```
Maude> red modelCheck(initial, [] (<> eats(0) /\ … /\ <> eats(4)), 'parity) .
rewrites: 558
result ModelCheckResult: counterexample(
  {< (o |0| o) ψ (o |1| o) ψ (o |2| o) ψ (o |3| o) ψ (o |4| o) ψ >,'left}
  {< (o |0| o) ψ (o |1| o) (ψ |2| o) ψ (o |3| o) ψ (o |4| o) ψ >,'left}
  {< (o |0| o) ψ (o |1| o) (ψ |2| o) ψ (o |3| o) (ψ |4| o) ψ >,'left}
  {< (ψ |0| o) ψ (o |1| o) (ψ |2| o) ψ (o |3| o) (ψ |4| o) >,'right}
  {< (ψ |0| o) ψ (o |1| o) (ψ |2| ψ) (o |3| o) (ψ |4| o) >,'release},
  {< (ψ |0| o) ψ (o |1| o) ψ (o |2| o) ψ (o |3| o) (ψ |4| o) >,'right}
  {< (ψ |0| o) ψ (o |1| ψ) (o |2| o) ψ (o |3| o) (ψ |4| o) >,'left}
  {< (ψ |0| o) (ψ |1| ψ) (o |2| o) ψ (o |3| o) (ψ |4| o) >,'release})
```

Not all problems of the counterexample above can be attributed to conflicts between philosophers. The only philosopher eating repeatedly in this trace is 1, but 3 and 4 could have eaten on their own, since they do not share any fork with 1. In fact, the strategy does not require the philosophers to eat whenever possible, although it can be modified to do it. Alternatively, a premise can be added to the LTL property $\Box \bigwedge_{k=1}^{4} \Diamond\, \texttt{used}(k)$ to ensure that no fork is underused. Anyhow, this does not prevent starvation.

```
Maude> red modelCheck(initial, [] (<> used(0) /\ … /\ <> used(4))
               -> [] (<> eats(0) /\ … /\ <> eats(4)), 'parity) .
rewrites: 4455
result ModelCheckResult: counterexample(…, …)
```

The omitted counterexample consists of eleven steps and shows that 0 and 3 do not eat because 1 and 2 are always *faster* to take their shared fork. In order to avoid starvation completely, an external synchronization source is required [Hoa85]. For example, a simple but perhaps too forced solution is establishing turns as in the `turns` strategy.

```
Maude> red modelCheck(initial, [] (<> eats(0) /\ … /\ <> eats(4)), 'turns) .
rewrites: 541
result Bool: true
```

Despite the nonterminating recursive definition of `turns`, the model checker terminates thanks to its ability to detect cycling tail-recursive calls even with arguments.

Branching-time properties can also be checked. For example, $\mathbf{A}\Box \bigwedge_{k=1}^{4} \mathbf{E}\Diamond$ `eats(k)` is a CTL formula claiming that philosophers never lose their chances to eventually take spaghetti. This property holds when deadlocks are prevented.

```
$ umaudemc check philosophers.maude initial
     'A [] (E <> eats(0) /\ … /\ E <> eats(4))' parity
The property is satisfied in the initial state
(58 system states, 6056 rewrites).
```

6.3 Processes and scheduling policies

The computers that we use in our everyday life are continuously running multiple interactive processes that share their resources [TB18]. Even if the number of physical and logical processors included in modern chips grows endlessly, the list of simultaneous processes increases too and the operating system has to decide which processes are granted access to the processing units at any given moment so that all tasks get done without unnecessary delay and degradation of the user experience. Moreover, these processes may depend on and communicate with each other and with external peripherals. Scheduling policies are strategies of the operating system to arrange the computer execution time. In this example, we will represent very simple instances of these in the Maude strategy language and check how properties are satisfied depending on them.

The simplified computer model used in this section is based on the Maude implementation of the Dekker algorithm in [CDE+07, EMS04]. It consists of a shared memory composed of integer cells indexed by the name of the variables, and a soup of processes running in the same processor.

```
sort Memory .
op [_,_] : Qid Int -> Memory [ctor] .
op none  : -> Memory [ctor] .
op __    : Memory Memory -> Memory [ctor assoc comm id: none] .

sorts Pid Process Soup MachineState .
subsort Process < Soup . subsort Int < Pid .
op [_,_]   : Pid Program -> Process [ctor] .
op empty   : -> Soup [ctor] .
op _|_     : Soup Soup -> Soup [ctor prec 61 assoc comm id: empty] .
op {_,_,_} : Soup Memory Pid -> MachineState [ctor] .
```

The third component of the machine state is the identifier of the last process to be run of sort `Pid`, which includes the integers as a subtype. Processes consist of a process identifier and a program in a simple imperative programming language:

```
sorts Test UserStatement Program .
subsort UserStatement < Program .
ops skip io       : -> Program [ctor] .
op _;_            : Program Program -> Program [ctor prec 61 assoc id: skip] .
op _:=_           : Qid Int -> Program [ctor] .
op _=_            : Qid Int -> Test [ctor] .
op if_then_fi     : Test Program → Program [ctor] .
op while_do_od    : Test Program → Program [ctor] .
op repeat_forever : Program → Program [ctor] .
```

The language constructs and their meaning are standard, where ; is sequential composition and := is assignment. Their semantics are defined by means of rules that manipulate the machine state. For instance, the rule for `repeat` is

```
vars I J : Pid .              var M : Memory .
vars P R : Program .          var S : Soup .

rl [exec] : {[I, repeat P forever ; R] | S, M, J}
        ⇒ {[I, P ; repeat P forever ; R] | S, M, I} .
```

The sort `UserStatement` may include other statements that are consumed when encountered, and the `io` instruction executes some input/output operation that is treated differently in the following.

```
var U : UserStatement .

rl [exec] : {[I, U ; R] | S, M, J} ⇒ {[I, R] | S, M, I} .
rl [io]   : {[I, io ; R] | S, M, J} ⇒ {[I, R] | S, M, I} .
```

Semaphores are also supported in the language with their two operations `wait` and `signal` implemented by the following rules:

```
ops wait signal : Qid → Program [ctor] .

var Q : Qid .                 var N : Int .

crl [exec] : {[I, wait(Q) ; R] | S, [Q, N] M, J}
          ⇒ {[I, R] | S, [Q, N - 1] M, I} if N > 0 .
rl  [exec] : {[I, signal(Q) ; R] | S, [Q, N] M, J}
          ⇒ {[I, R] | S, [Q, N + 1] M, J} .
```

The rule for `wait` fails if the memory value `N` in `Q` is not greater than zero, and so processes in that situation will not advance.

The following programs are written using this language: they execute a critical section (`crit` is defined as a user statement) protected by a semaphore in the `mutex` variable.

```
eq p = repeat
         wait('mutex) ;
         crit ;
         signal('mutex)
       forever .
```

```
eq pIo = repeat
           wait('mutex) ;
           crit ;
           signal('mutex) ;
           io
         forever .
```

The `pIo` program additionally executes an input/output operation outside the critical section.

In the rewriting system described above, the `exec` rule tries to run any process in the soup nondeterministically, and so their execution is completely concurrent. Even so, semaphores are enough to guarantee that only one process is in the critical section at the same time. To check this, we define an atomic proposition `inCrit(`k`)` that tells whether the process k is in the critical section, extending as usual the system module.

```
subsort MachineState < State .
var MS : MachineState .
eq {[I, crit ; R] | S, M, J} ⊨ inCrit(I) = true .
eq MS ⊨ inCrit(I) = false [owise] .
```

Moreover, since the property claiming that no pair of processes are simultaneously in the critical section

$$\Box\neg\left(\bigvee_{m=0}^{N}\bigvee_{n=0}^{m-1}\texttt{inCrit}(n)\wedge\texttt{inCrit}(m)\right)$$

is verbose and depends on the number N of processors, we built it equationally together with the initial configuration.

```
op onlyOne       : Nat         → Formula .
op inCritFormula : Nat Nat     → Formula .
op initial       : Nat Program → MachineState .

vars N M : Nat .             var P : Program .
eq onlyOne(N) = []~ inCritFormula(N, N) .
eq inCritFormula(0, 0) = False .
eq inCritFormula(1, s(M)) = inCritFormula(M, M) .
eq inCritFormula(s(N), M) = (inCrit(N) /\ inCrit(M))
                          \/ inCritFormula(N, M) [owise] .

eq initial(N, P) = { initialSoup(N, P), ['mutex, 1], 0 } .
eq initialSoup(0, P) = empty .
eq initialSoup(s(N), P) = initialSoup(N, P) | [s(N), P] .
```

The mutual exclusion property specified above can be checked with the standard model checker for any fixed number of processors.

```
Maude> red modelCheck(initial(4, p), onlyOne(4)) .
reduce in CS-SCHECK : modelCheck(initial(4, p), onlyOne(4)) .
rewrites: 4373
result Bool: true
```

However, it is not true that every process eventually gets into the critical section (we omit the counterexample because it has 39 states).

```
Maude> red modelCheck(initial(4, p), <> inCrit(1)) .
reduce in CS-SCHECK : modelCheck(initial(4, p), onlyOne(4)) .
rewrites: 201
result ModelCheckerResult: counterexample(..., ...)
```

On top of this specification and in a separate strategy module, we have defined different scheduling policies as strategies. Since changing the active process involves the expensive operation of saving or restoring its execution context, operating systems try to amortize it by executing as many instructions as possible before changing again. The `blocked` policy keeps executing the current process `P` with `exec[I ← P]`, where the identifier `P` has been obtained with the **`matchrew`** from the machine state. However, if this process is blocked by an `io` operation or in a semaphore, the rule `exec` executes any other process nondeterministically.

```
sd blocked := ((matchrew MS s.t. {S, M, P} := MS
                 by MS using exec[I ← P]) or-else (try(io) ; exec))
               ; blocked .
```

Another common scheduling policy is called *round-robin*. The `roundRobin` strategy maintains in its argument a list of process identifiers and tries to execute them cyclically, passing to the next state when the current one gets blocked. The process list can be initially empty or incomplete, in case it is filled nondeterministically with the available processes.

```
sd roundRobin(nil) := matchrew MS s.t. {[P, R] | S, M, J} := MS
                        by MS using (exec[I ← P] ; roundRobin(P)) .
sd roundRobin(P LP) := exec[I ← P] ? roundRobin(P LP) : (
  try(io) ;
  ((matchrew MS s.t. {[I, R] | S, M, J} := MS /\ not(occurs(I, P LP))
         by MS using exec[I ← I])
  ? (matchrew MS s.t. {S, M, I} := MS by MS using roundRobin(I LP P))
  : roundRobin(LP P))
```

However, a process can still occupy the processor forever. The round-robin policy can be modified to be *preemptive* by assigning a maximum time slice for each process and pass the usage of the processor to the next one once it is consumed, if it was not blocked before.

```
sd roundRobin(P LP, 0, N) := try(io) ; (
  (matchrew MS s.t. {[I, R] | S, M, J} := MS /\ not(occurs(I, P LP))
         by MS using exec[I ← I])
  ? (matchrew MS s.t. {S, M, I} := MS by MS using roundRobin(I LP P, N, N))
  : roundRobin(LP P, N, N)) .

sd roundRobin(P LP, s(K), N) := exec[I ← P] ?
     roundRobin(P LP, K, N) : roundRobin(P LP, 0, N) .
```

Since the uncontrolled model already protects the critical section, and because all linear-time properties satisfied by a given model are satisfied in the same model under the control of any strategy, the critical section will always be protected. However, other fairness properties may depend on the scheduling policy. For instance, the property $\Box\Diamond$ `inCrit(1)` is not satisfied neither with the `blocked` policy nor with `roundRobin`.

```
Maude> red modelCheck(initial(4, p), [] <> inCrit(1), 'blocked) .
rewrites: 7
result ModelCheckerResult: counterexample(..., ...)
```

However, the counterexample consists only of 5 states instead of the 39 obtained with the standard model checker, and they are easier to understand in that they obey the restrictions of the strategy. They both show a process executing its loop continuously because it is never blocked. The preemptive version of round-robin makes the property hold.

```
$ umaudemc check semaphore.maude 'initial(4, p)' '[] <> inCrit(1)' \
           'roundRobin(nil, 5, 5)'
The property is satisfied (1621 system states, 37735 rewrites).
```

We have used the `umaudemc` interface, since it allows calling `roundRobin` with arguments without declaring a new strategy. A time slice of 5 has been fixed and the initial process list is empty. These parameters are immaterial to the satisfaction of the property since all processes are identical, but their values may affect the size of the model. We could have fixed the process order with the strategy `roundRobin(1 2 3 4, 5, 5)` instead, and the model would only have 90 states.

Replacing the `p` program by `pIo`, which includes a blocking input/output operation, changes the situation. Thanks to the blocking operation, the `roundRobin` strategy is enough to ensure fairness since no process is left with the monopoly on the processor.

```
$ umaudemc check semaphore.maude 'initial(4, pIo)' '[] <> inCrit(1)' \
           'roundRobin(nil)'
The property is satisfied (705 system states, 6199 rewrites).
```

However, the `blocked` policy may never activate a given process, because the next to obtain the processor when the active process is blocked is chosen nondeterministically. We obtain a counterexample where processes 2, 3, and 4 are being executed in turns repeatedly.

```
$ umaudemc check semaphore.maude 'initial(4, pIo)' '[] <> inCrit(1)' blocked
The property is not satisfied (18 system states, 73 rewrites).
[...]
```

Finally, we try a last strategy `roundRobin*` where the turn of each process is defined by an iteration of `exec`. Under the usual semantics of the iteration in the strategy language, a process could extend its turn indefinitely without releasing the processor for the others. However, under the Kleene-star semantics of Section 3.6.1, the iteration means that processes can execute an arbitrary but always finite number of instructions in each turn, i.e. that they will eventually release the processor. This is a typical fairness constraint that has been incorporated to the strategy itself.

```
sd roundRobin*(P LP) := (exec[I <- P] ? exec[I <- P] * ; amatch ['mutex, s(N)]
                                          : idle) ; roundRobin*(LP P) .
```

For simplicity, we have removed in this definition the generation and completion of the process list, assuming that the order is fixed by the caller. The `amatch` test prevents a process from releasing the processor when it has the mutex, so that at least one of the other processes can advance in the round. The fairness property that has already been checked with the previous strategies can be verified for `roundRobin*` using the model checker presented in Section 4.5.

```
$ umaudemc check semaphore.maude initial '[] <> inCrit(1)' \
           'roundRobin*(1 2 3 4)' --kleene-iteration
The property is satisfied (194 system states, 116695 rewrites, 2 Büchi states)
```

Notice that we have included the `--kleene-iteration` flag to enable this feature, since otherwise the property would not be satisfied.

```
$ umaudemc check semaphore.maude initial '[] <> inCrit(1)' 'roundRobin*(1 2 3 4)'
The property is not satisfied
(19 system states, 66 rewrites, 2 Büchi states)
```

This example could be expanded to support more realistic models and scheduling policies.

6.4 The RIP protocol

This section describes a simple application of the strategy language to the specification of a communication protocol. The *Routing Information Protocol* [Mal98] is an interior gateway protocol for the interchange of routing information based on distance vectors.

Internet is composed of different interconnected networks. A message between two hosts in two different networks may travel through a sequence of adjacent networks. The devices that connect them are the *routers*, which decide where to send the data packages so that they arrive to their destinations, preferably by the shortest path. In order to do their job, they need to know the topology of the interconnected networks. This information can be manually established by the system administrator as a table associating a network or arrangement of networks either to the physical interface of the router directly connected to that network or to the next router towards the destination. Such association is called a *routing table*, and this approach, *static routing*. Instead, routing tables can be constructed dynamically and be aware of network changes, based on data shared by the routers themselves. This is called *dynamic routing*, and RIP is one of its first representative protocols.

RIPv2 routers exchange their routing tables periodically with their neighbors. When using the IP protocol, each table entry locally describes the next step to reach an aggregation of networks given by an IP address and a mask, here represented in *Classless Inter-Domain Routing* notation.[2] The other relevant fields are the IP address of the next-step router in case the network is not directly reachable, the physical interface that must be used to reach the destination, and the distance, which usually measures the number of router jumps and is called *hop count*.

```
op <_,_,_,_,_> : CIDR IPAddr Interface Nat Nat -> Route [ctor] .
```

[2] IPv4 addresses are 32-bit words. Networks (or aggregations of them) are collections of IP addresses with a common prefix, whose extension is indicated by the mask. CIDR identifies a network by an address followed by the prefix length.

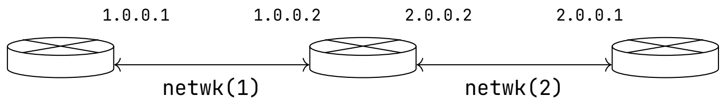


Figure 6.3: A simple network topology.

RIP considers hop-counts above 15 as infinity, so that networks at a greater distance are unreachable. Moreover, the table includes an invalidation timer, to discard entries when no information about them has been received for a significant amount of time.

In this example, the whole network specification is object-oriented [CDE+20; §8]. The objects are the routers, whose attributes are a routing table and an interface list that enumerates their IP addresses for each directly connected network.

```
op Router : -> Cid [ctor] .                              *** class identifier

op table:_ : RouteTable -> Attribute [ctor gather (&)] .
op interfaces:_ : Interfaces -> Attribute [ctor gather (&)] .

op none : -> Interfaces [ctor] .
op _▷_ : NetworkId CIDR -> Interfaces [ctor prec 31] .
op __ : Interfaces Interfaces -> Interfaces [ctor assoc comm id: none] .
```

Networks are not represented by the structure of the configuration, but by means of unique identifiers of sort `NetworkId`. Since broadcast and multicast messages used by this protocol are distributed inside the boundaries of a network, this information will be relevant. Every IP message contains a sender and a receiver IP address, and a payload. Multicast and broadcast messages additionally include a network identifier.

```
op IPMessage : IPAddr IPAddr Payload -> IPMsg [msg] .
op IPMessage : IPAddr NetworkId IPAddr Payload -> IPMsg [msg] .
```

In the case of RIP messages, the payload consists of some fields described in the protocol specification, whose main part is the exchanged routing table. The actual definitions of all the previous elements and some operations required to manipulate them are specified in various functional and system modules (`IP-ADDR`, `IP-MESSAGES`, `RIP-ENTRIES`, `ROUTER`... in the example source).

The basic operation of the RIP protocol is the interchange of routing tables. This can be triggered as a response to an initial request, or by the detection of changes in the network and the routing tables. However, we will focus on the *gratuitous responses* that routers send approximately every 30 seconds to all their interfaces. These messages are multicast to all the routers, and strategies will be used to deliver them once to all the interested receivers. But first, the router actions are described as rules in a system module `RIP`. For example, the emission of a (gratuitous) response is specified by the following rule, where the message address 224.0.0.9 is a multicast address recognized by all RIPv2 routers.

```
rl [response] :
   < I : Router | table: RT, interfaces: NId ▷ A / Mask Ifs, Attrs >
=> < I : Router | table: RT, interfaces: NId ▷ A / Mask Ifs, Attrs >
   IPMessage(224 . 0 . 0 . 9, NId, A, ripMsg(2, 2, export(RT))) .
```

This simple rule will not be admissible in a specification without strategies, because it can be endlessly applied. Its counterpart is the rule in charge of processing the RIP message and updating the routing tables accordingly:

```
crl [readResponse] :
   < I : Router | table: RT, interfaces: NId ▷ A / Mask Ifs, Attrs >
   IPMessage(224 . 0 . 0 . 9, NId, O, ripMsg(2, 2, RE))
⇒ < I : Router | table: import(O, NId, RE, RT),
                 interfaces: NId ▷ A / Mask Ifs, Attrs >
   IPMessage(224 . 0 . 0 . 9, NId, O, ripMsg(2, 2, RE))
if O =/= A .
```

Notice that the message is not removed from the configuration, since it is a multicast message. The strategy will remove it when all the routers have received it, using the `remove-message` rule. Time progresses using the rule `update-timer` that increments the internal timers of the routers.

Over these rules, the protocol operation on a time window of 30 seconds is specified in the `iteration` strategy. The strategy has two other overloads, in which it delegates more specific tasks.

```
strat iteration                  @ Configuration .
strat iteration : Set{Oid}       @ Configuration .
strat iteration : Oid Interfaces @ Configuration .

sd iteration := matchrew C by C using (
                    one(update-tick(allOids(C))) ;
                    iteration(allOids(C))
                ) .

op allOids : Configuration → Set{Oid} .

eq allOids(none) = empty .
eq allOids(< I : Class:Cid | Attrs:AttributeSet > C) = I, allOids(C) .
eq allOids(M:Msg C) = allOids(C) .
```

The parameterless `iteration` strategy collects with the `allOids` function all the object identifiers of the configuration `C` captured by `matchrew`, and passes them to the second overload. Before that, the timers of the routers are updated by the strategy `update-tick`, which executes the rule `update-timer` once per router.

```
sd iteration(empty) := handleResponse ! .
sd iteration((I, IS)) := try(
    matchrew C s.t. < I : Router | table: RT, interfaces: Ifs > D := C
        by C using one(iteration(I, Ifs)) ;
    handleResponse *
  ) ;
  iteration(IS) .
```

Using that set of identifiers, the `iteration(Set{Oid})` overload iterates over all the objects in the configuration. Each object is matched against a `Router` object pattern, where the identifier `I` is the same as the `I` in the strategy argument. If the matching succeeds, the third overload of `iteration` is called to send responses from every interface of the router. An undetermined number of these responses or of those previously sent by other routers and not yet delivered will be received using the `handleResponse` strategy. These two strategies will be explained later, but we already see that the emission and reception of responses can be freely interleaved. Otherwise, if the matching does not succeed, the object identifier does not designate a router and the strategy fails inside the `try`, jumping to the recursive call to `iteration` for `IS`. Since the argument is a set, in each strategy call the matches will be multiple, and `I` will be bound to every element in the set. Hence, objects will be visited nondeterministically in any possible order, and the strategy only enforces that they are visited only once in each rewriting path. This is convenient because the order in which the routers issue their response is not fixed, and different outcomes could be obtained with different orders. The third overload of `iteration` sends responses by invoking the `response` rule for the router `I` on each interface `NId`.

```
sd iteration(I, none) := idle .
sd iteration(I, NId ▷ N Ifs) := response[I ← I, NId ← NId] ;
                                one(iteration(I, Ifs)) .
```

Unlike the previous strategies, how `iteration` visits the interfaces is irrelevant because the table is not updated meanwhile. Thus, every such call is surrounded by a `one` to avoid unnecessary computation.

As we have anticipated, some response messages may remain in the configuration and others may be processed before the next object is addressed due to the `handleResponse *` expression in the `iteration` strategy over routers. At the end, when the set of identifiers is `empty`, all the remaining messages are processed using the normalization operator. The `handleResponse` strategy takes any multicast message in the soup and delivers it to all its recipients.

```
strat handleResponse                               @ Configuration .
strat handleResponse : NetworkId IPAddr Payload Oid @ Configuration .

sd handleResponse := matchrew C s.t. IPMessage(A, NId, O, P) D := C
  by C using (one(handleResponse(NId, O, P, allOids(C))) ;
              remove-message[A ← A, NId ← NId, O ← O, P ← P]) .

sd handleResponse(NId, O, P, empty) := idle .
sd handleResponse(NId, O, P, (I, IS)) :=
  try(readResponse[I ← I, NId ← NId, O ← O, P ← P]) ;
  one(handleResponse(NId, O, P, IS)) .
```

It is assumed that all the messages are received simultaneously by all the routers, and so `one` is used. The definition of this strategy follows the same scheme of the `iteration` overloads, making each object accept the selected message by fixing the variables in the `readResponse` rule when applying it.

As an example, let us apply an iteration on the network of Figure 6.3, where each router has its table filled with the networks to which it is connected, namely 1.0.0.0/8 and 2.0.0.0/8.

```
Maude> srew linear using iteration .

Solution 1
rewrites: 136244 in 699ms cpu (699ms real) (194856 rewrites/second)
result Configuration: < r1 : Router | table:
    < 1 . 0 . 0 . 0 / 8, 0 . 0 . 0 . 0, netwk(1), 0, 0 >
    < 2 . 0 . 0 . 0 / 8, 1 . 0 . 0 . 2, netwk(1), 1, 0 >,
    interfaces: netwk(0) ▷ 1 . 0 . 0 . 1 / 8 >
  < r2 : Router | table:
    < 1 . 0 . 0 . 0 / 8, 0 . 0 . 0 . 0, netwk(1), 0, 0 >
    < 2 . 0 . 0 . 0 / 8, 0 . 0 . 0 . 0, netwk(2), 0, 0 >,
    interfaces: netwk(0) ▷ 1 . 0 . 0 . 2 / 8
                netwk(1) ▷ 2 . 0 . 0 . 2 / 8 >
  < r3 : Router | table:
    < 1 . 0 . 0 . 0 / 8, 2 . 0 . 0 . 2, netwk(2), 1, 0 >
    < 2 . 0 . 0 . 0 / 8, 0 . 0 . 0 . 0, netwk(2), 0, 0 >,
    interfaces: netwk(1) ▷ 2 . 0 . 0 . 1 / 8 >

No more solutions.
rewrites: 136244 in 699ms cpu (699ms real) (194856 rewrites/second)
```

Since this network is too small, an iteration is enough to propagate all the routing information, no matter in which order the responses occur. If a similar linear topology is designed with four routers, two iterations are needed to complete the routing information of the routers at both

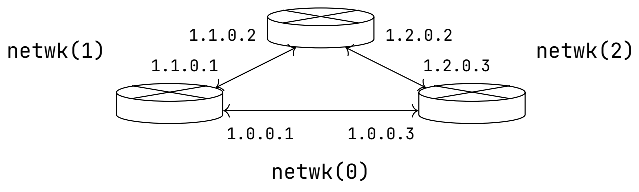


Figure 6.4: A triangular network.

ends, and multiple results are obtained in the first iteration. Other configurations may lead to unwanted situations due to known faults of the protocol like the count to infinity problem. Techniques to mitigate them like the *divided horizon* and *poisoned response* [Mal98] have also been specified and they can be seen in the example collection, but we do not describe them here because strategies are not specially relevant in their implementation.

6.4.1 Model checking

Using the specification of the RIP protocol and the strategy-aware model checker, we will check some temporal properties on small network examples. As already done in previous sections, the Maude specification should be extended with the required atomic propositions in a separate module, say `RIP-PREDS`.

```
mod RIP-PREDS is
  protecting RIP .
  including SATISFACTION .

  var N : CIDR .     vars H T : Nat .          var NId : NetworkId .
  var A : IPAddr .   var  Ifs : Interfaces .   var C   : Configuration .
  var I : Oid .

  op countReaches : Nat → Prop [ctor] .
  op reachable : Oid CIDR → Prop [ctor] .

  eq < I : Router | table: < N, A, NId, H, T > RT, interfaces: Ifs > C
       ⊨ countReaches(H) = true .
  eq C ⊨ countReaches(H) = false [owise] .

  ceq C ⊨ reachable(I, N) = true
   if < N, A, NId, H, T > := tableEntry(C, I, N) /\ H < 16 .
  eq C ⊨ reachable(I, N) = false [owise] .
endm
```

The `countReaches(H)` proposition holds when a router has the hop count `H` in some entry of its routing table, and `reachable(I, N)` tells whether the network `N` is reachable for the router `I`. The auxiliary function `tableEntry(C, I, N)` is defined equationally to obtain the table entry for the network `N` in the router `I` of the configuration `C`, or `noRoute` if it does not exist. Even if a router has an entry for a network in its table, that network may not be reachable if the hop count is the RIP infinity, 16.

Using these atomic propositions, we will check that the leftmost router `r1` and the rightmost router `r3` of the linear network of Figure 6.3 are mutually reachable at some point and since then. In the same formula, we also check that the hop counts of all routers are at most one.

```
Maude> red modelCheck(linear, <> [] (reachable(r1, 2 . 0 . 0 . 0 / 8) /\
  reachable(r3, 1 . 0 . 0 . 0 / 8)) /\ [] ~ countReaches(2), 'iterateForever) .
```

```
StrategyModelCheckerSymbol: Examined 13239 system states.
rewrites: 161411 in 652ms cpu (654ms real) (247325 rewrites/second)
result Bool: true
```

We have used to control the system a new strategy `iterateForever` that honors its name:

```
strat interateForever @ Configuration .
sd iterateForever := iteration ? iterateForever : idle .
```

However, the stabilization of the routing information is achieved in the first iteration, and this can be checked with the model checker using the next ◯ operator and making the strategy `iteration` an atomic step of the model.

```
Maude> red modelCheck(linear, O [] (reachable(r1, 2 . 0 . 0 . 0 / 8)
  /\ reachable(r3, 1 . 0 . 0 . 0 / 8))
  /\ [] ~ countReaches(2), 'iterateForever, 'iteration) .
StrategyModelCheckerSymbol: Examined 2 system states.
rewrites: 289282 in 1371ms cpu (1372ms real) (210863 rewrites/second)
result Bool: true
```

In this case, the number of system states has decreased significantly, because the intermediate states inside the window of an iteration are hidden. However, the amount of rewrites has increased since the detection of execution cycles is not as effective when using opaque strategies. In the triangular network of Figure 6.4, we can also check that every router can eventually and forever since reach the network to which it is not directly connected.

```
Maude> red modelCheck(loop, <> [] (reachable(r1, 1 . 2 . 0 . 0 / 16)
  /\ reachable(r2, 1 . 0 . 0 . 0 / 16) /\ reachable(r3, 1 . 1 . 0 . 0 / 16))
  /\ [] ~ countReaches(2), 'iterateForever) .
StrategyModelCheckerSymbol: Examined 142117 system states.
rewrites: 2368601 in 9588ms cpu (9596ms real) (247018 rewrites/second)
result Bool: true
```

Since there are more connected networks and more messages, the size of the model for this network topology is greater. This network also remains stable after the first iteration, and it can be checked as in the previous case.

In the first lines of this example, we introduce RIP as a dynamic routing protocol that adapts to the changes in the network, but the situations we have model checked in the previous paragraphs are essentially static. In order to analyze more interesting cases, let us insert the `break-link` rule that disconnects a router from an interface `NId`. The corresponding table entry in the routing table is also removed.

```
crl [break-link] :
  < I : Router | table: < N, 0 . 0 . 0 . 0, NId, 0, T > RT,
                 interfaces: NId ▷ A / Mask Ifs >
=>
  < I : Router | table: RT, interfaces: Ifs >
if N = normalize(A / Mask) .
```

In the linear topology, we will check whether the connection between `r1` and `r3` is preserved after disconnecting the middle router `r2` from the right network. Instead of using the Maude interpreter, we will check the property using the more flexible `umaudemc` interface.

```
$ umaudemc check RIP.maude linear '<> [] reachable(r1, 2 . 0 . 0 . 0 / 8)' \
  'iteration ; break-link[I <- r2, NId <- netwk(2)] ; iterateForever' \
  --opaque iteration --elabel '%l' \
  --slabel '{tableEntry(%t, r1, 2 . 0 . 0 . 0 / 8)}
           {tableEntry(%t, r2, 2 . 0 . 0 . 0 / 8)}'
```

```
The property is not satisfied in the initial state
(19 system states, 533160 rewrites, 2 Büchi states)
| noRoute
| < 2 . 0 . 0 . 0 / 8,0 . 0 . 0 . 0,netwk(2),0,0 >
v  iteration
| < 2 . 0 . 0 . 0 / 8,1 . 0 . 0 . 2,netwk(1),1,0 >
| < 2 . 0 . 0 . 0 / 8,0 . 0 . 0 . 0,netwk(2),0,0 >
v  break-link
| < 2 . 0 . 0 . 0 / 8,1 . 0 . 0 . 2,netwk(1),1,0 >
| noRoute
v  iteration
| < 2 . 0 . 0 . 0 / 8,1 . 0 . 0 . 2,netwk(1),1,1 >
| < 2 . 0 . 0 . 0 / 8,1 . 0 . 0 . 1,netwk(1),2,0 >
v  iteration
| < 2 . 0 . 0 . 0 / 8,1 . 0 . 0 . 2,netwk(1),3,0 >
| < 2 . 0 . 0 . 0 / 8,1 . 0 . 0 . 1,netwk(1),2,0 >
v  iteration
...
v  iteration
| | < 2 . 0 . 0 . 0 / 8,1 . 0 . 0 . 2,netwk(1),16,0 >
| | < 2 . 0 . 0 . 0 / 8,1 . 0 . 0 . 2,netwk(1),16,0 >
| v  iteration
< v
```

The strategy expression that controls the system executes an iteration, then breaks the mentioned link, and continues iterating forever. Instead of the full configuration, the states of the counterexample show the route table entries for the network 2.0.0.0/8 in both `r1` and `r2`, using the `--slabel` option of `umaudemc` and the `tableEntry` function. The verification result is the expected one, because the connection between those networks has been effectively interrupted. The counterexample shows the well-known *count-to-infinity* problem. In the first iteration, `r1` learns the route to the right network from a response sent by `r2`, then `r2` loses its route to that network when the link is broken, but the gratuitous responses of `r1` make it believe that `r3` is still reachable through `r1`. However, `r1` knows that it has learned this route from `r2` so it keeps increasing the hop count when it receives responses from the middle router, and so both routers start a mutual count to infinity. When the infinity value is reached, both routers realized that the right network is not reachable. If the *divided horizon* technique had been used, `r1` would not have sent the route to `r3` to the router it has learned it from, so `r2` would have not recovered the entry for the right network and `r1` would have eventually lost its entry because of the timers.

A similar property can be checked on the `loop` network, but in this case it is satisfied, because there is an alternative route to the disconnected network.

```
$ umaudemc check RIP.maude loop '<> [] reachable(r1, 1 . 2 . 0 . 0 / 16)' \
  'iteration ; break-link[I <- r2, NId <- netwk(1)] ; iterateForever'
The property is satisfied in the initial state
(1426910 system states, 22146343 rewrites, 2 Büchi states)
```

6.5 Basic completion

In equational logic, given a set E of equations, the *word problem* is deciding whether two given terms t_1 and t_2 satisfy $t_1 =_E t_2$. Even though the problem is undecidable, incomplete procedures can be constructed for a wide range of instances. A *completion procedure* [BN98; §7] is a method to transform a set of equations into a convergent (confluent and terminating) rewriting

$$\begin{array}{llll}
\text{Deduce} \quad \dfrac{\langle E, R\rangle}{\langle E \cup \{s = t\}, R\rangle} \quad \text{if } s \leftarrow_R u \rightarrow_R t & & \text{Simplify} \quad \dfrac{\langle E \cup \{s = t\}, R\rangle}{\langle E \cup \{u = t\}, R\rangle} \quad \text{if } s \rightarrow_R^+ u \\
\text{Orient} \quad \dfrac{\langle E \cup \{s = t\}, R\rangle}{\langle E, R \cup \{s \rightarrow t\}\rangle} \quad \text{if } s > t & & \text{R-Simplify} \quad \dfrac{\langle E, R \cup \{s \rightarrow t\}\rangle}{\langle E, R \cup \{s \rightarrow u\}\rangle} \quad \text{if } t \rightarrow_R^+ u \\
\text{Delete} \quad \dfrac{\langle E \cup \{s = s\}, R\rangle}{\langle E, R\rangle} & & \text{L-Simplify} \quad \dfrac{\langle E, R \cup \{s \rightarrow t\}\rangle}{\langle E \cup \{u = t\}, R\rangle} \quad \text{if } s \rightarrow_R^{\sqsupset} u
\end{array}$$

Figure 6.5: Bachmair and Dershowitz's inference rules.

system. Whenever it succeeds, provable E-equalities can be decided by exhaustively reducing both sides using the calculated rules, and comparing their canonical forms syntactically.

Many concrete completion procedures can be expressed as particular ways of applying a set of inference rules (see Figure 6.5) proposed by Bachmair and Dershowitz [BD94], in other words, as specific strategies in that deduction system. Lescanne [Les90] described various such procedures as a combination of *transition rules* + *control*, and implemented them in CAML. Completion procedures were also implemented in ELAN [KM95], at the Maude metalevel [CM97], and even using the Maude strategy language [VM11]. However, the separation between rules and control is not clearly enforced in these works, since the rules are adapted to the specific data structure of each method. In this section, we propose an improvement of [VM11] in which a single fixed set of rules is used to express the different specific procedures as stateful strategies. These strategies are specified in separate strategy modules on top of the same system module `COMPLETION`, as shown in Figure 6.6.

The basic state of the completion procedures is a pair (E, R) of equations and rules. The procedure begins with a set of equations $(E, \varnothing)$, and concludes with a set of rules $(\varnothing, R)$ unless it fails in finite time or it loops forever. Equations are oriented to become rules and critical pairs are calculated during the process. The syntax for equations, rules, the calculation of critical pairs, and the rest of the infrastructure are defined in the functional module `CRITICAL-PAIRS`. Since each example of equational system to be completed has its own term signature and since a precedence order on its function symbols is required by the completion procedures, almost every module in Figure 6.6 is parameterized by the `MODULE` theory or by its extension `MODULE-AND-ORDER`. Using the order relation $\gg$ on the function symbols of the instance, a lexicographic path ordering $>$ will be derived for all the terms. The most basic definitions are those for equations `Eq`, rules `Rl`, and sets of these:

```
op _=._ : Term Term -> Eq [comm prec 60] .
op _->_ : Term Term -> Rl [prec 60] .

op mtEqS : -> EqS [ctor] .   op mtRlS : -> RlS [ctor] .
```

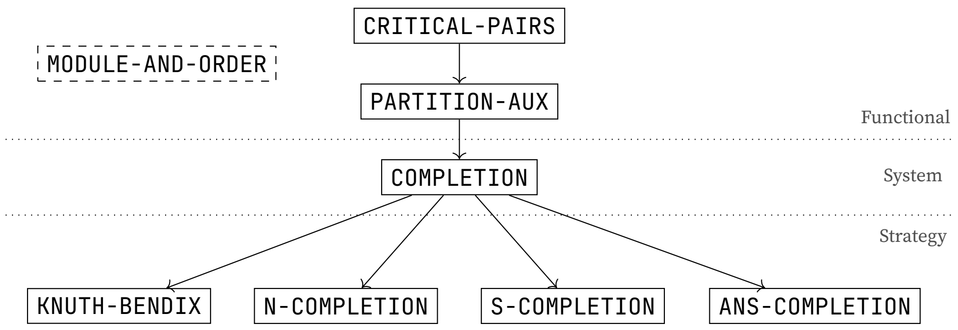


Figure 6.6: Module structure of the basic completion procedures.

```
op __ : EqS EqS → EqS [assoc comm id: mtEqS prec 70] .
op __ : RlS RlS → RlS [assoc comm id: mtRlS prec 70] .
```

Bachmair and Dershowitz's inference rules and the state pair on which they operate are defined in the parameterized system module `COMPLETION`:[3]

```
mod COMPLETION{X :: MODULE-AND-ORDER} is
  pr CRITICAL-PAIRS{ForgetOrder}{X} .

  sort System .
  op <_,_> : EqS RlS → System [ctor] .

  var  E      : Eqs .
  vars R QR   : RlS .
  vars s t u  : Term .

  *** [...] (the completion inference rules as rewrite rules)
endm
```

First, the rule `Orient` takes an equation $s \approx t$ and orients it as $s \to t$ whenever $s > t$.

```
crl [Orient] : < E s =. t , R >
            ⇒ < E, R s → t > if s > t .
```

Since the operator `=.` is commutative, the equation sides are interchangeable. Trivial equations are removed by `Delete`, and `Simplify` reduces any side of an equation by exhaustively rewriting it with the already generated rules. This is implemented, using Maude metalevel facilities, by the `reduce` function imported from `CRITICAL-PAIRS`.

```
rl [Delete] : < E s =. s, R >
           ⇒ < E, R > .

crl [Simplify] : < E s =. t, R >
              ⇒ < E u =. t, R >
  if u := reduce(s, R) .
```

For their part, rules are simplified using the `R-Simplify` and `L-Simplify` rules, which simplify the left and righthand side of a rule respectively. Observe that there are two sets in the rule pattern, and the term is only reduced by the rules in `R`, while the set `QR` of *quiet rules* is not used. The different completion procedures could select which rules to reduce with by fixing `R` and `QR` conveniently.

```
crl [R-Simplify] : < E, R QR s → t >
                ⇒ < E, R QR s → u >
  if u := reduce(t, R) .

crl [L-Simplify] : < E, R QR s → t >
                ⇒ < E u =. t, R QR >
  if u := reduce>(s → t, R) .
```

Since reductions are decreasing with respect to the term order, the righthand side simplification still satisfies $s > u$. However, the order may not be preserved with `L-Simplify`, so the rule is removed and reinserted as an equation. Moreover, the lefthand side term s is not reduced exhaustively by all the rules in R as before, but it is applied a single arbitrary rule $l \to r$ in R whose lefthand side l cannot be reduced by $s \to t$. This is the meaning of the $\to_R^{\sqsupset}$ arrow in Figure 6.5, and the behavior of the `reduce>` function. Finally, `Deduce` infers new identities by

[3]Since the module `CRITICAL-PAIRS` does not depend on the order, it is parameterized only by `MODULE`, and the `ForgetOrder` view from `MODULE` to `MODULE-AND-ORDER` allows instantiating it by the parameter `X`.

calculating the critical pairs of a rule `r` with a selected subset of rules `R`. Here, we have also included a set `QR` to allow restricting the rules whose critical pairs are calculated.

```
crl [Deduce] : < E, R QR >
            ⇒ < E equations(critical-pairs(r, R)), R QR >
if r R' := R QR .
```

Applying the inference rules carelessly does not always lead to a terminating and confluent set of rules, even if one exists. The second entry of the state is always a terminating rewriting system because of the strict order on the terms, but it may not be confluent if some critical pairs have not been calculated. Moreover, the deduction system itself is not terminating, since generating critical pairs leading to known identities may always be possible. Thus, strategies must be used to control the rule execution.

In the following, we describe the Knuth-Bendix completion and N-completion procedures. The slightly more complex S-completion and ANS-completion procedures are available in the example collection.

6.5.1 Knuth-Bendix completion

The Knuth-Bendix [KB70] completion procedure follows the script below for a state (E, R):

1. Select an equation from E. If there is none, stop with success.
2. Simplify its both sides using the rules in R.
3. If both sides of the equation become identical, remove the equation and go back to step 1. Otherwise, continue to step 4.
4. Orient the equation and add the resulting rule to R. If this is not possible, stop with a failure.
5. Calculate the critical pairs between the new rule and the whole rule set, and add them as equations to E. Go back to step 1.

The procedure can be specified by the following `compl` strategy, which uses `deduction` as an auxiliary strategy. It consists of a `matchrew` combinator, whose content implements the five steps of the script, inside a normalization operator `!` that iterates them until they fail.

```
sd compl := (matchrew Sys s.t. < E s =. t, R > := Sys by
               Sys using (try(Simplify[E ← E, s ← s]) ;
                          try(Simplify[E ← E, s ← t]) ;
                          (Delete[E ← E] or-else
                             (Orient[E ← E] ;
                              deduction(R))
                          )
                         )
            ) ! .
```

First, the `matchrew` patterns select an equation from the state. If there is none, the matching will fail and the strategy execution will stop due to the normalizing operator. Otherwise, the equation is simplified (step 2) using the `Simplify` rule. By fixing the variable `E` in the lefthand side `< E s =. t, R >` of the rule, the simplification is applied only to the selected equation. Moreover, setting `s` in the rule to `s` and `t` alternatively in the strategy context ensures that both sides are reduced, and since the simplification rule fails when the term is already simplified, these rule applications are surrounded by a `try` operator. Then, the strategy attempts to `Delete` the equation (step 3). On success, the substrategy finishes successfully and another iteration is started because of the normalizing operator. On failure, the `or-else` alternative is executed.

The rule `Orient` orients the selected equation unless it is not orientable (step 4). In this case, the application fails, and so the iteration loop and the whole procedure stops. As there is still an equation in the state, the failure can be observed in the procedure's output. On success, the `deduction` strategy is called with the set of rules `R` from the matching, which contains all rules except the new one.

```
sd deduction(R) := matchrew Sys s.t. < E, R s ⟶ t > := Sys by Sys using
                      Deduce[r ⟵ s ⟶ t, QR ⟵ mtRlS] .
```

In order to generate the critical pairs between the new rule and the whole rule set, the `deduction` strategy definition applies `Deduce` with its variable `r` instantiated to that rule and `QR` to the empty set. The new rule `s ⟶ t` is recovered in the `matchrew` by matching against `< E, R s ⟶ t >` where `R` is instantiated to its value from the context, which is the argument of the strategy that contained all the rules except the new one. Using this procedure, all the critical pairs are generated for every rule, so whenever the strategy terminates with an empty equation set, the resulting set of rules is a convergent rewriting system [Hue81].

As a simple example to illustrate the procedure, consider a signature with two unary symbols f and g, whose precedence is $f \gg g$, and an equation $f(f(x)) = g(x)$. The algorithm is run by:

```
Maude> srew < 'f['f['x:S]] =. 'g['x:S], mtRlS > using compl .

Solution 1
rewrites: 6943 in 6ms cpu (6ms real) (1048000 rewrites/second)
result System: < mtEqS, 'f['f['x:S]] ⟶ 'g['x:S]
                        'f['g['x:S]] ⟶ 'g['f['x:S]] >

No more solutions.
rewrites: 6943 in 6ms cpu (6ms real) (1048000 rewrites/second)
```

The algorithm has successfully finished, providing a confluent term rewriting system that solves the word problem for this equational system.

6.5.2 N-completion

N-completion refines the previous procedure by the simplification of the rules and a more efficient computation of critical pairs. These rules are partitioned into a set T of rules whose critical pairs have not been calculated yet, and its complement $R \setminus T$ (*marked rules* in Huet's terminology [Hue81]). Since this partition should be part of the procedure state, in the original implementation [VM11] the state pair was extended to a triple and the inference rules of the completion procedure were changed as a consequence. Here, the information will be held in the strategy parameters:

```
strats N-COMP simplify-eqs @ System .
strats N-COMP success orient deduce simplify-rules : RlS @ System .

sd N-COMP    := N-COMP(mtRlS) .
sd N-COMP(T) := success(T) or-else (match < mtEqS, R > ? deduce(T)
                                                        : orient(T)) .
```

N-completion is implemented by a collection of strategies that take a set of rules as argument, which stands for the set T mentioned before, a subset of rules of the state term. The entry point is the parameterless overload of `N-COMP` that calls its homonym strategy with an empty T set. During execution, T is updated and passed on as an argument among the different mutually recursive strategies. According to the `N-COMP` definition, the procedure tries to execute one of the auxiliary strategies `success`, `deduce`, and `orient`. The first one tests whether the procedure has successfully finished, which is exactly when there are neither pending equations nor rules whose critical pairs have not yet been calculated.

```
sd success(mtRlS) := match < mtEqS, R > .
```

Remember that a call that does not match any definition is a failure, so when T is non-empty this strategy fails. In that case, the procedure continues either with the `deduce` or with the `orient` strategy depending on whether the equation set is empty or not. Hence, equations are greedily oriented and added to T by `orient`, and only when there are no equations, the critical pairs are calculated by `deduce`. This strategy nondeterministically takes a rule `r` from T, deduces the identities from its critical pairs with respect to the rest of the rules, and then calls `simplify-rules`. Since the critical pairs for `r` have just been calculated, `r` is no longer included in the strategy argument. The definition is only executed if T is non-empty, but this is an invariant at this point because otherwise `success` would have finished the execution before.

```
sd deduce(r T) := Deduce[r ← r, QR ← mtRlS] ;
                  simplify-rules(T) .

sd simplify-rules(T) := matchrew Sys s.t. < E, R > := Sys by Sys using (
          (L-Simplify[QR ← mtRlS] | R-Simplify[QR ← mtRlS])
          ? matchrew Sys' s.t. < E', R' > := Sys' by Sys' using
              simplify-rules(combine(T, R, R'))) :
          : N-COMP(T)
        ) .
```

The `simplify-rules` strategy tries to simplify either the left or righthand side of any rule (inside or outside T) using the rest of the rules (for that reason, the quiet rules variable `QR` is set to the empty set), and calls itself recursively when it succeeds to make the simplification exhaustive. If no more simplifications are possible, a recursive call to `N-COMP` continues the procedure. `L-Simplify` and `R-Simplify` can modify the rule or convert it into an equation, so we should track the changes to update T. Using the two nested `matchrew`, we probe the rule set before and after the simplification, to compare them and find out what changed. The new T is calculated by a function `combine` defined in `PARTITION-AUX` by some simple equations.

$$\text{combine}(T, R, R') = \begin{cases} T & r \notin T \\ (T \setminus (R \setminus R')) \cup R' \setminus R & r \in T \end{cases}$$

The third auxiliary strategy, `orient`, is in charge of simplifying and orienting equations. The `matchrew` is added a condition `s > t` to know which is the rule that `Orient` would add, so that it can be included in the set T. Incidentally, this condition reduces the number of distinct matches of the subterm operator due to the commutativity of equation symbols.

```
sd orient(T) := simplify-eqs ;
                (match < mtEqS, R >
                  ? N-COMP(T)
                  : matchrew Sys s.t. < s =. t E, R > := Sys
                    /\ s > t by Sys using (
                      Orient[E ← E] ;
                      N-COMP(s → t T)
                    )
                ) .

sd simplify-eqs := (Delete | Simplify) ! .
```

N-completion is more efficient than the initial procedure, and so fewer rewrites are required to compute the same completion of the previous section.

```
Maude> srew < 'f['f['x:S]] =. 'g['x:S], mtRlS > using N-COMP .
```

```
Solution 1
rewrites: 2279 in 3ms cpu (3ms real) (691654 rewrites/second)
result System: < mtEqS, 'f['f['x:S]] → 'g['x:S]
                        'f['g['x:S]] → 'g['f['x:S]] >

No more solutions.
rewrites: 2279 in 3ms cpu (3ms real) (691654 rewrites/second)
```

In order to show that the efficiency improvement allows calculating the completion of more complex systems in reasonable time, we will apply it to the group axioms $e * x = x$, $I(x) * x = e$ and $(x * y) * z = x * (y * z)$, where $*$ is the binary group operation, I is the inverse, and e the identity element, whose precedence is set to $I \gg * \gg e$. The previous Knuth-Bendix algorithm does not terminate in hours for this problem, but N-completion quickly finds a solution using the depth-first search of the `dsrewrite` command.

```
Maude> dsrew [1] < '*['e.S ,'x:S] =. 'x:S  '*['I['x:S], 'x:S] =. 'e.S
                   '*['*['x:S, 'y:S], 'z:S] =. '*['x:S, '*['y:S, 'z:S]],
                   mtRlS > using N-COMP .

Solution 1
rewrites: 222369 in 309ms cpu (311ms real) (718093 rewrites/second)
result System: < mtEqS, '*['e.S,'x:S] → 'x:S
                        '*['x3:S,'*['I['x3:S],'z5:S]] → 'z5:S
                        '*['x3:S,'I['x3:S]] → 'e.S
                        '*['z2:S,'e.S] → 'z2:S
                        '*['*['x:S,'y:S],'z:S] → '*['x:S,'*['y:S,'z:S]]
                        '*['I['x:S],'x:S] → 'e.S
                        '*['I['y1:S],'*['y1:S,'z1:S]] → 'z1:S
                        'I['e.S] → 'e.S
                        'I['*['x3:S,'y5:S]] → '*['I['y5:S],'I['x3:S]]
                        'I['I['y1:S]] → 'y1:S >
```

Using the calculated rewriting system (`result` in the following), we can check whether the term $I(x*(y*z))$ is equal to $(I(z)*I(y))*I(x)$ by reducing both terms to normal form and comparing the results syntactically.

```
Maude> red reduce('I['*['x:S, '*['y:S, 'z:S]]], result) .
rewrites: 21 in 3ms cpu (2ms real) (6300 rewrites/second)
result Term: '*['I['z:S],'*['I['y:S],'I['x:S]]]

Maude> red reduce('*['*['I['z:S], 'I['y:S]], 'I['x:S]], result) .
rewrites: 19 in 0ms cpu (2ms real) (~ rewrites/second)
result Term: '*['I['z:S],'*['I['y:S],'I['x:S]]]
```

The strategies for the more complex S-completion and ANS-completion procedures follow the same principles, but their strategies receive more parameters standing for finer rule partitions. All the details can be found in the `completion` directory of the example collection.

6.6 The 15-puzzle

In this section, we come back to the 15-puzzle introduced in Section 3.1, and show a strategy to solve it. Remember that the game, which dates from the 1870s, consists of fifteen tiles numbered from 1 to 15 lying in a framed square surface of side length 4. The blank left by the absent sixteenth tile can be used to move the numbers from their positions. Given any arrangement of the puzzle, the goal is to put the tiles as in Figure 3.1, in ascending order from left to right and from the top to the bottom, with the blank at the last position. However, only half of the initial settings can be solved because there exists a relation between the parity of the permutation and the position of the blank, invariant by the allowed moves [Luc92; §8]. The puzzle can be generalized to any side length n with $n^2 - 1$ tiles.

In order to solve the puzzle, a search can be executed with the `search` command or with the strategy commands `srewrite` and `dsrewrite` using the strategy below. However, only puzzles that are near the solution can be expected to be solved like this, since the search space size is impractical for an unguided search, as it has approximately $2.1 \cdot 10^{13}$ states.

```
Maude> search [1] puzzle1 =>* P:Puzzle s.t. P:Puzzle = solved .

Solution 1 (state 571)
states: 572
rewrites: 20713 in 20ms cpu (23ms real) (1035650 rewrites/second)
P:Puzzle --> *** solved

Maude> srew [1] puzzle1 using (left | right | up | down) * ;
                          match P:Puzzle s.t. P:Puzzle = solved .

Solution 1
rewrites: 52906 in 72ms cpu (71ms real) (734805 rewrites/second)
result Puzzle: *** solved
```

where

```
eq puzzle1 =  b   6   2   4  ;      eq solved =  1   2   3   4  ;
              1   5   3   8  ;                   5   6   7   8  ;
              9  10   7  11  ;                   9  10  11  12  ;
             13  14  15  12  .                  13  14  15   b  .
```

In fact, this problem is a classic example used to illustrate search heuristics and algorithms like A*. Using this approach and the strategy-parameterized branch-and-bound implementation described in [RMP+19c], we have written an instance of this problem that is available in the example collection of the strategy language. Although finding (even the length of) the shortest sequence of moves that leads to the solution is NP-complete [RW90], non-optimal solutions can be found in polynomial time with deterministic algorithms not based on search. Next, we describe a strategy that solves the puzzle in $\mathcal{O}(n^6)$ moves, where n is the side length, and is inspired by the resolution method discussed in [Luc92]. The strategy can be clearly improved in several ways, but we want to keep it as simple as possible.

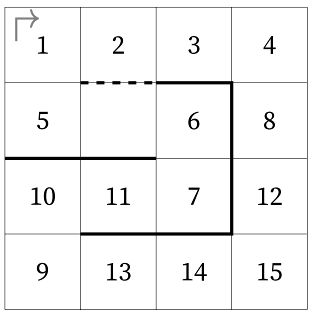


Figure 6.7: A circuit in the 15-puzzle.

The key fact is that moving the tiles in a closed circuit through the board does not change the relative position of the numbers within it. Figure 6.7 depicts a possible circuit, where the

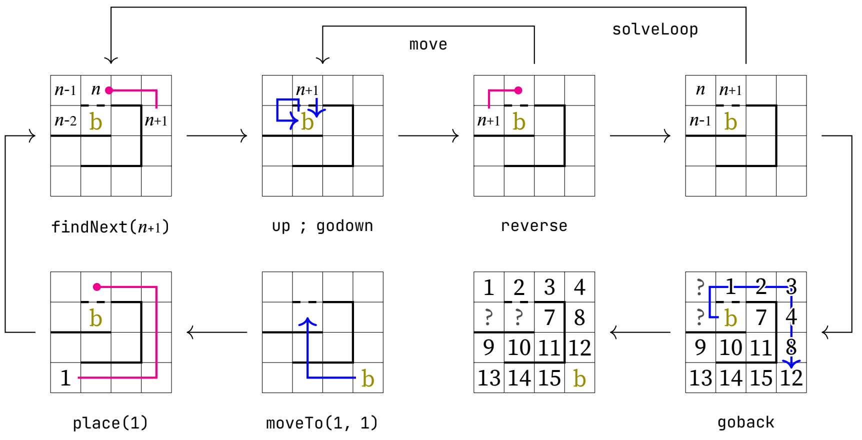


Figure 6.8: Overview of the `solve` strategy.

thick lines represent barriers that cannot be crossed. The act of moving the blank through the circuit is called *rotating* and does not alter the sequence of numbers (1, 2, 3, 4, 8, 12, 15, 14, 13, 9, 10, 11, 7, 6, 5) that can be read clockwise from 1 in the figure. However, all tiles are shifted one position in the direction opposite to the movement of the blank. Using successive rotations, we can place any number above or below the dashed line, and change its relative order in the sequence by slipping it across that line. Like a sorting algorithm, every element could be moved to its correct position in the sequence, except that tiles cannot be freely swapped but only jump over pairs of contiguous tiles.

Assuming the blank is initially below the dashed line, these rotations are described in Maude by the strategies `rotate` and `reverse`, which move the blank in the direction indicated by the arrow and in the reverse one respectively. Remember that numbers get moved in the opposite direction.

```
strats rotate reverse godown goup goback @ Puzzle .

sd rotate := left ; up ; right ; right ; right ;
             down ; down ; down ; left ; left ; left ;
             up ; right ; right ; up ; left .
```

Their definitions are explicit concatenations of rules that follow immediately from Figure 6.7. Other auxiliary strategies are defined similarly: always without crossing any thick line, the strategies `goup` and `godown` move the blank between the positions below and above the dashed line, and `goback` puts it in the lower-right corner, its desired final position. Their complete specifications are available in the example collection.

The solving strategy is called `solve`. Its first action is moving the blank below the dashed line with `moveTo`, which delegates on `move` from Section 3.1. Tile 1 is then placed above the line by successive rotations, and the *sorting loop* in `solveLoop` starts. At the end of this loop, the tiles in the circuit will be properly ordered if possible, the strategy will place 1 above the dashed line again, and execute `goback`, which moves the blank to the lower-right corner, leaving 1 and the other tiles in their wanted positions. A schema of the procedure is depicted in Figure 6.8.

```
strat  solve                  @ Puzzle .
strats place solveLoop : Tile @ Puzzle .

sd solve := moveTo(1, 1) ; place(1) ; solveLoop(1) ;
            place(1) ; goback .

sd place(T) := (match P s.t. T =/= atPos(P, 1, 0) ; rotate) ! .
```

The solveLoop strategy iterates on the expected tile sequence, obtained manually from the circuit, sorting the numbers in the board to match it. The precondition is that the input parameter T is above the dashed line and preceded by the correct prefix LL in the circuit, and the purpose is to make the next expected tile NT occupy the next position. NT is found in the board by findNext using successive rotations until the expected tile is above the dashed line, as place did. Its second parameter counts the distance, which is finally passed on to move to displace it back as many times as indicated by this number.

```
vars LL LR : Row . vars T NextT Pen Last : Tile .

csd solveLoop(T) := rotate ; findNext(NextT, 0) ; solveLoop(NextT)
 if LL T NextT LR Pen Last := sequence .

csd solveLoop(T) := idle if LL T Pen Last := sequence .

strat findNext : Tile Nat @ Puzzle .
strat move     : Nat      @ Puzzle .

sd findNext(T, N) := match P s.t. T = atPos(P, 1, 0) ? move(N)
    : (rotate ; findNext(T, s(N))) .
```

The definitions of solveLoop are conditional to allow obtaining the next tile from the sequence by matching.

The move strategy is defined recursively. In case the distance is greater than two, the tile is moved down across the dashed line with up, so that it advances two positions against the rotation direction and towards its expected position. Here, an invariant is that the current tile is above the line and the blank is below. To maintain it for the recursive call, godown is called and two reverse rotations make the current tile recover its previous position.

```
sd move(0)       := idle .
sd move(1)       := rotate ; goup ; down ; reverse ; reverse .
sd move(s(s(N))) := up ; godown ; reverse ; reverse ; move(N) .
```

This method does not work when the distance is one, but since the tile that occupies the desired position is misplaced, it can be moved clockwise with a similar operation that makes the tile going up across the dashed line instead. This movement is harmless because it moves the tile to the unordered portion of the sequence. To see that the moved tile does not surpass the first element of the sequence, observe that the last two elements of the list are omitted in solveLoop when the sequence is matched to LL T NT LR Pen Last. Obviously, nothing should be done for the last element, but neither for the penultimate, which can never be swapped with its neighbor as follows from the parity analysis of the puzzle. If the last two elements are misplaced, the problem is not solvable.

Now, we can execute the strategy to solve some puzzles:

```
Maude> srew (5 1 4 8 ; 2 14 15 3 ; 9 7 6 11 ; 13 10 b 12) using solve .

Solution 1
rewrites: 28868 in 32ms cpu (31ms real) (902937 rewrites/second)
result Puzzle: 1        2       3       4 ;
               5        6       7       8 ;
               9        10      11      12 ;
               13       14      15      b

No more solutions.
rewrites: 28868 in 32ms cpu (31ms real) (902937 rewrites/second)
```

If the starting puzzle is unsolvable, this is revealed by the position of tiles 5 and 6 being swapped.

```
Maude> srew (15 2 1 12 ; 8 5 6 11 ; 4 9 10 7 ; 3 14 13 b) using solve .

Solution 1
rewrites: 40568 in 40ms cpu (41ms real) (1014200 rewrites/second)
result Puzzle: 1        2        3        4 ;
               6        5        7        8 ;
               9        10       11       12 ;
               13       14       15       b

No more solutions.
rewrites: 40568 in 40ms cpu (41ms real) (1014200 rewrites/second)
```

When we introduced this example in Chapter 3, we admitted that the data representation was not intended to be efficient but easily readable, to illustrate the strategy language at work. However, we can give the board a more efficient representation as a set of position-to-content pairs:

```
op [_,_,_] : Nat Nat Tile → Puzzle [ctor] .
op __ : Puzzle Puzzle → Puzzle [ctor assoc comm id: empty] .
```

Like this, the rules `up` and `down` can be implemented by simpler unconditional rules:

```
rl [up]   : [X, Y, T] [X, s(Y), b] ⇒ [X, Y, b] [X, s(Y), T] .
rl [down] : [X, Y, b] [X, s(Y), T] ⇒ [X, Y, T] [X, s(Y), b] .
```

And since the strategies above are built solely on the rule names and the function `atPos`, the strategies do not need to be modified to work with the new representation, in which the problem is solved using fewer rewrites.

```
Maude> srew [0,0,5] [0,1,2] ... [2,3,b] ... [3,3,12] using solve .

Solution 1
rewrites: 1630 in 20ms cpu (21ms real) (81500 rewrites/second)
result Puzzle: [0,0,1] [0,1,5] [0,2,9]  [0,3,13]
               [1,0,2] [1,1,6] [1,2,10] [1,3,14]
               [2,0,3] [2,1,7] [2,2,11] [2,3,15]
               [3,0,4] [3,1,8] [3,2,12] [3,3,b]

No more solutions.
rewrites: 1630 in 20ms cpu (21ms real) (81500 rewrites/second)
```

Chapter 7

Extensions of the strategy language

The reflective generation and manipulation of strategies is the common factor of the following examples and extensions of the strategy language. Reflection plays an important role in Maude, for testing new features, implementing verification tools and metalinguistic interfaces. We have applied this tradition to the strategy language to implement some interesting features. All these extensions have been done while preserving the object-level interaction that facilitate the usage of strategies, and the availability of the strategy-aware model checker.

The first introductory example addresses normalization in context-sensitive rewriting using strategies generated according to the term signature. This theory-dependent transformation is generalized in the next section to extend the Maude strategy language with new combinators from ELAN and Stratego that are not available in Maude, as seen in Section 3.8. The proposed skeleton for these transformations is generic and applicable to other extensions. However, it has some limitations that can be avoided with an executable specification of the small-step operational semantics of the strategy language in Section 3.5, which is described in the next section. As a kind of bootstrapping, this example could further clarify the connection of this semantics with model checking, be used to represent more exotic strategy combinators, and provide the basis of the prototype implementation of the Kleene-star model checker in Section 4.5. Finally, multistrategies are introduced to define the control of a system with multiple strategies for different actors, players, or parts instead of a single one. Executing and model checking these systems is possible, as shown with an example of the tic-tac-toe game.

Most of these examples have been presented at the workshop WPTE 2020 [RMP+20a], and this chapter is based on the journal paper [RMP+22a]. The extensible executable semantics of the strategy language is included in [RMP+21a] in a shorter form.

7.1 Context-sensitive rewriting

Context-sensitive rewriting [Luc20] is a restricted form of term rewriting where simple constraints attached to the symbols of the signature exclude some of their arguments from being rewritten. Maude has builtin support for this kind of restrictions by means of the `strat` and `frozen` attributes.

op f `:` $s_1 \cdots s_n \to s$ `[strat(`$i_1 \cdots i_k$ `0) frozen(`$j_1 \cdots j_l$`)] .`

Regarding equational reduction, the evaluation strategy attribute `strat` specifies a zero-terminated list of argument indices $i_m \in \{1, \ldots, n\}$ that fix the order in which arguments are reduced before applying equations to the top, while absent arguments are not reduced at all. By default, the evaluation strategy is `1 2` $\cdots$ n `0`. Regarding rules, the `frozen` attribute inhibits rewriting with rules inside a given subset of arguments $j_m \in \{1, \ldots, n\}$. These restrictions may prevent nonterminating evaluations, but their direct application is not enough to obtain irreducible terms, for which strategies are needed, as we will see with a lazy programming example. Generating these strategies from the context-sensitive restrictions is the purpose of our introductory metalevel transformation. Let us present first the following functional module [DEL04] that attempts to specify a lazy list of integers:

```
fmod LAZY-LIST is
  protecting INT .
  sort LazyList .

  op nil : -> LazyList [ctor] .
  op _:_ : Int LazyList -> LazyList [ctor] .

  var E : Int . var N : Nat . var L : LazyList .

  op take : Nat LazyList -> LazyList .
  eq take(0, L) = nil .
  eq take(s(N), E : L) = E : take(N, L) .

  op natsFrom : Nat -> LazyList .
  eq natsFrom(N) = N : natsFrom(N + 1) .
endfm
```

Even though `natsFrom`(n) represents an infinite list, containing all natural numbers from n, we would expect that the lazy evaluation of a term like `take(3, natsFrom(0))` leads to `0:1:2:nil`. However, Maude's `reduce` command eagerly applies equations in an innermost leftmost manner, so the evaluation of this term will not terminate because of the continuous reduction of the tails in the `natsFrom` definition. Fortunately, the Maude `strat` attribute can be used on the `_:_` operator to avoid reducing the tail of the list, by changing its `[ctor]` attribute to `[ctor strat(1 0)]`. However, this context-sensitive restriction limits rewriting too much, and no valid result is still produced:

```
Maude> reduce take(3, natsFrom(0)) .
rewrites: 2
result LazyList: 0 : take(2, natsFrom(0 + 1))
```

In the vocabulary of context-sensitive rewriting, `strat` and `frozen` annotations correspond to *replacement maps* $\mu : \Sigma \to \mathcal{P}(\mathbb{N})$ where $\mu(f) \subseteq \{1, \dots, \text{arity}(f)\}$ for all $f \in \Sigma$. Reduction is only allowed in the *μ-replacing positions* of any term, defined recursively as

$$\text{Pos}^{\mu}(f(t_1, \dots, t_n)) = \{\varepsilon\} \cup \bigcup_{i \in \mu(f)} \{i\}\, \text{Pos}^{\mu}(t_i),$$

where ε denotes the top position and the word wi the i-th argument of the subterm at position w. Exhaustively reducing in these positions yields *μ-normal forms*, exactly what the previous command did for `take(3, natsFrom(0))` and $\mu(\_:\_) = \{1\}$. As that execution shows, μ-normal forms are not necessarily normal forms of the unrestricted rewrite system, but μ-normalization can be useful to build complete and lazy normalization procedures [DEL04]. Normalization can be achieved via μ-normalization using a *layered evaluation* that safely resumes reduction on the subterms of non-replacing positions [Luc20; §9.3], as illustrated in Figure 7.1 for the term `take(2, natsFrom(1))`. At each step, the highlighted subterms at a given level of the term tree are applied μ-normalization, and this continues to its arguments down to the leaves. This procedure is implemented by means of a generated strategy proposed by Salvador Lucas [Luc21] that needs to traverse the term. Since the Maude strategy language does not offer any resource to do it generically, a signature-aware strategy must be produced.

The following function `csrTransform` implements a metalevel module transformation that extends the metarepresentation of the input module `M` with strategy declarations and definitions that normalize terms as described in the previous paragraph. Its global shape is given by the

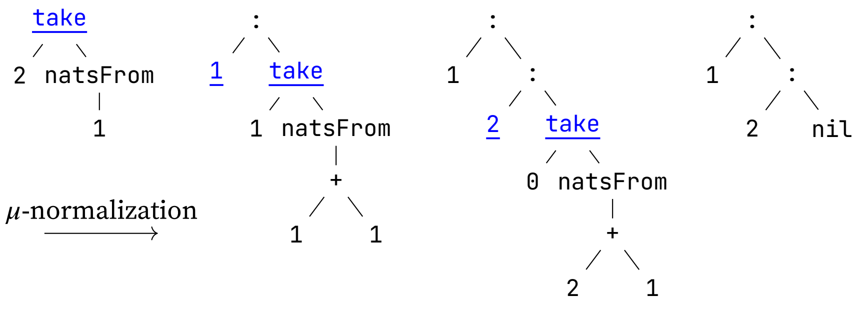


Figure 7.1: Layered normalization of `take(2, natsFrom(1))`.

following equation.[1]

```
op csrTransform : Module -> StratModule .
eq csrTransform(M) = smod append(getName(M), 'CSR)  is
  getImports(M)                                     *** module importation
  sorts 'AnyTerm ; getSorts(M) .                            *** sort decls
  getSubsorts(M)                                         *** subsort decls
  strat2frozen(getOps(M))                               *** operator decls
  getMbs(M)                                      *** sort membership axioms
  none                                                       *** equations
  getRls(M)                                                      *** rules
  eqs2rls(getEqs(M))
  getStrats(M)                                          *** strategy decls
  (strat 'norm-via-munorm : nil @ 'AnyTerm [none] .)
  (strat 'munorm : nil @ 'AnyTerm [none] .)
  (strat 'decomp : nil @ 'AnyTerm [none] .)
  getSds(M)                                       *** strategy definitions
  (sd 'norm-via-munorm[[empty]] :=
       'munorm[[empty]] ; 'decomp[[empty]] [none] .)
  (sd 'munorm[[empty]] := one(all) ! [none] .)
  (sd 'decomp[[empty]] := makeDecomp(getOps(M)) [none] .)
endsm .
```

Except for the new strategies, the transformed module is essentially a copy of the original one. However, since the Maude strategy language can only control rule application, we translate all equations into rules,[2] and all `strat` attributes to `frozen` annotations.

```
eq eqs2rls(none) = none .
eq eqs2rls(eq L = R [Attrs] . Eqs) = rl L => R [Attrs] . eqs2rls(Eqs) .
eq eqs2rls(ceq L =  R if C [Attrs] . Eqs) =
           crl L => R if C [Attrs] . eqs2rls(Eqs) .
```

The transformed module is always a strategy module, regardless of which type of module `M` is, where three strategies are declared. The entry point of the layered normalization procedure is `norm-via-munorm`, which executes two auxiliary strategies `munorm` for μ-normalization, and then `decomp` for resuming normalization inside frozen arguments. `munorm` is implemented by exhaustively (`!`) applying the rules in the module respecting the `frozen` restrictions (**`all`**). Assuming the input system is μ-confluent, i.e. confluent under the context-sensitive restrictions,

[1]Strategy declarations must include the intended sort to which they are applied after the `@` sign. However, the strategies defined here are somehow polymorphic, so we declare `AnyTerm` just to take its place.

[2]Respecting the `owise` semantics would require specifying strategies to apply their translation as rules likewise, but for simplicity we assume that there are no `owise` annotations.

the order in which rules are applied does not affect the result, so **all** is executed for efficiency under the **one** operator that discards alternative rewrite orders. For its part, the `decomp` strategy continues normalization on the symbol arguments. Strategies can be applied inside subterms in the Maude strategy language using the **matchrew** combinator, so one is generated for each $f \in \Sigma$ to apply `norm-via-munorm` recursively on every subterm:

matchrew $f(x_1, \ldots, x_n)$ **by** …, x_i **using** `norm-via-munorm`, …

The decomposition strategy `decomp` is defined as the disjunction of all these combinators. When applied on a particular term, only the **matchrew** for the top symbol of that term will match. Since this definition depends on the signature of the module, `decomp` is reflectively generated by the `makeDecomp` function that walks through the operators declared in the module.

```
var Q  : Qid .    var Ops   : OpDeclSet .    var N : Nat .
var Ty : Type .   var NeTyL : NeTypeList .

op makeDecomp : OpDeclSet -> Strategy .
eq makeDecomp(none) = fail .
eq makeDecomp(op Q : nil -> Ty [Attrs] . Ops) =
  (match qid(string(Q) + "." + string(Ty)) s.t. nil) | makeDecomp(Ops) .
eq makeDecomp(op Q : NeTyL -> Ty [Attrs] . Ops) =
  (matchrew Q[makeVarList(NeTyL, 1)] s.t. nil by makeUsingPart(NeTyL, 1))
  | makeDecomp(Ops) .
```

Constants (operators with an empty list `nil` of arguments) do not have arguments in which normalization should be resumed, but they must also be matched by the `decomp` strategy so that it does not fail when any of them is encountered. In this case, instead of a **matchrew**, a test **match** is used with the metarepresentation of that constant as described in Section 2.5.2. Auxiliary functions like `makeVar` and `makeVarList` are used to generate sequentially-numbered variable metarepresentations of the given sorts for the **matchrew** pattern. These variables are mapped to the `norm-via-munorm` strategy by the `makeUsingPart` function.

```
op makeUsingPart : NeTypeList Nat -> UsingPairSet .
eq makeUsingPart(Ty, N) = makeVar(N, Ty) using 'norm-via-munorm[[empty]] .
eq makeUsingPart(Ty NeTyL, N) = makeUsingPart(Ty, N),
                                makeUsingPart(NeTyL, s(N)) .

op makeVar : Nat Type -> Variable .
eq makeVar(N, Ty) = qid("X" + string(N, 10) + ":" + string(Ty)) .
```

Finally, the term `csrTransform(upModule('LAZY-LIST, true))` can be reduced to obtain the transformed `'LAZY-LIST` module. Then, the `norm-via-munorm` strategy can be applied to a term using the `metaSrewrite` function, whose inputs and results are written at the metalevel:

```
red metaSrewrite(csrTransform(upModule('LAZY-LIST, true)),
                 'take['s_^3['0.Zero], 'natsFrom['0.Zero]],
                 'norm-via-munorm[[empty]],
                 breadthFirst, 0) .
rewrites: 3129
result ResultPair: {'_:_['0.Zero,'_:_['s_['0.Zero],
            '_:_['s_^2['0.Zero],'nil.LazyList]]],'LazyList}
```

Alternatively, Full Maude can be used with terms and strategies at the object level:[3]

```
(select CSR-TRANSFORM .)
(load csrTransform(upModule('LAZY-LIST, true)) .)
```

[3]Full Maude commands are typed between parentheses, once the `full-maude.maude` file is loaded.

```
(select LAZY-LIST-CSR .)
(srewrite take(3, natsFrom(0)) using norm-via-munorm .)

Solution 1
result LazyList: 0 : 1 : 2 : nil

No more solutions
```

The evaluation now terminates with a meaningful result. When the rewrite system is confluent and terminating under the restrictions, as in this case, any search strategy, `breadthFirst` or `depthFirst`, `srewrite` or `dsrewrite`, would produce the same result since there is a single solution and a finite state space. The `norm-via-munorm` strategy can be used to normalize any term in the module without computing the transformation again. However, since it explicitly refers to the signature of that module, its definition is not applicable to any other module.

The following section shows an extension of the Maude strategy language adding new combinators based on the subject module in which strategies are to be applied. Using that extended language the `norm-via-munorm` strategy will be defined directly and more succinctly.

7.2 Theory-dependent extensions of the strategy language

When the Maude strategy language was designed, the objective was not to include a vast repertory of operators to concisely express a wide range of tasks, like in the case of Stratego [BKV+08], but to be compact and expressive enough. Thanks to reflection, the language can be extended to better suit a specific purpose or to incorporate a missing feature. In this section, we apply this principle and describe a general schema to construct strategy language extensions by some module transformations without losing any of its advantages from the user point of view, like the interaction at the object level and with the strategy-aware model checker. As an example, the language will be extended with the so-called *congruence operators* and *generic traversals*, both available in Stratego and the first also in ELAN, as discussed in Section 3.8. Both operator families depend on the signature of the subject module where strategies are applied, so a module transformation is required to implement them.

The procedure we will follow is directly applicable to other extensions, and it is supported by a collection of helper modules that save work and avoid writing boilerplate code for each extension. It consists of the following steps illustrated in Figure 7.2:

1. Extending the `META-LEVEL` with the metarepresentation of the new strategy combinators, probably depending on the module M where strategies are to be applied.

2. Since the builtin `metaSrewrite` does not support the new combinators and in order to execute them, extended expressions are translated to the standard language by extending the skeleton of a `transform` function. Moreover, the translation allows using the strategy-aware model checker on extended strategies and all other strategy-related machinery of the interpreter for free.

3. In order to write extended strategies at the object-level, an extensible grammar of the strategy language `SLANG-GRAMMAR` can be added productions for the new operators. Strategies are parsed as terms with the builtin `metaParse` function, and transformed to their metarepresentations by an extensible `stratParse` function.

4. A parameterized interactive Maude interface is provided to operate with extended strategies completely at the object-level. It admits extended strategies in its `srewrite` command and in the strategy definitions of strategy modules. A `umaudemc`-based program for model checking with these strategies and modules is available too.

$$\begin{tikzcd}
\texttt{metaSrewrite} & \overline{\beta} & \boxed{\texttt{META-LEVEL}} \arrow[d] & & \boxed{\texttt{SLANG-GRAMMAR}} \arrow[d] & \\
& & \boxed{ext(M)} & \boxed{M} \arrow[l, dashed] \arrow[r, dashed] & \boxed{extGram(M)} & \\
& & \overline{\alpha} \arrow[uul, "\text{transform}"] & & t \arrow[ll, "\text{stratParse}"'] & \texttt{metaParse}
\end{tikzcd}$$

Figure 7.2: Typical structure of a strategy language extension.

As a result, extended strategy expressions can be used almost anywhere an original expression could have been used, although not directly in the commands of the Maude interpreter. Since strategies are translated to the standard strategy language, some extensions are not easily implemented using this approach, and the extensible executable semantics of the language in Section 7.3 will be more useful for them.

The two families of operators mentioned at the beginning of this section will let us illustrate the proposed procedure. Remember from Section 3.8 that congruence operators $f(\alpha_1, \ldots, \alpha_n)$ are equivalent to a `matchrew` of the form

$$f(\alpha_1, \ldots, \alpha_n) \equiv \texttt{matchrew } f(x_1, \ldots, x_n) \texttt{ by } x_1 \texttt{ using } \alpha_1, \ldots, x_n \texttt{ using } \alpha_n.$$

Generic traversals apply a strategy to all or some of the arguments of a term regardless of its top symbol. They are `gt-all(`α`)` that applies α to all arguments of the top symbol, `gt-one(`α`)` that applies α to the first argument from left to right in which it succeeds, and `gt-some(`α`)`, which is equivalent to `test(gt-any(`α`)) ; gt-all(try(`α`))`.[4]

Congruence operators. First, we extend the metalevel with a homonym `Strategy` symbol for each data constructor of M taking as many `Strategy` arguments as the arity of the original one:

```
op generateCongOps : OpDeclSet -> OpDeclSet .
eq generateCongOps(none) = none .
eq generateCongOps(op Q : TyL -> Ty [ctor Attrs] . Ops) =
  (op Q : repeatType('Strategy, size(TyL)) -> 'Strategy [ctor removeId(Attrs)] .)
  generateCongOps(Ops) .
eq generateCongOps(Op Ops) = generateCongOps(Ops) [owise] .
```

The auxiliary function `repeatType` builds a list with the given number of repetitions of its first argument, and `removeId` removes identity axiom attributes of the original operators, which are meaningless in the congruence operators. Overloaded symbols may produce different conflicting declarations if their attributes do not coincide, so generated operators are given a second pass to remove potential conflicts.

Extended strategies can now be expressed at the metalevel. However, the builtin descent function `metaSrewrite` is unaware of these new operators and we do not want to implement the strategy language from scratch, we should translate them to the standard subset. This translation is defined in the extended `META-LEVEL` as a function `transform` between terms of sort `Strategy`. The complete recursive definition of this function would be large and repetitive, so an extensible and generic one is supplied to facilitate the task of defining extensions. This is provided by the `SLANG-EXTENSION-STATIC` module to be included in the transformed module, where equations are given for the standard constructors and the user only has to provide equations for the new elements.

```
fmod SLANG-EXTENSION-STATIC is
```

[4]The original names of generic traversal operators in Stratego do not include the `gt-` prefix, which is used here to avoid confusion with the `one` and `all` operators of the Maude strategy language.

```
  protecting META-LEVEL .

  op transform : Strategy Nat -> Strategy .

  var N : Nat . var S : Strategy .
  eq transform(idle, N) = idle .
  eq transform(top(S), N) = top(transform(S, N)) .
  ...
endfm
```

The `transform` operator takes a natural number as a second argument, used as an index to generate fresh variables in nested **`matchrew`**s, since their bindings are permanent. Some helper functions like `makeVar` and `makeConstant` that already appear in the previous example are included in the module too. Note that the equations defining `transform` are generated by the module transformation, so they must metarepresent `Strategy` terms and involve two levels of reflection.

```
op generateCongOpsDefs : OpDeclSet -> EquationSet .

eq generateCongOpsDefs(none) = none .
eq generateCongOpsDefs(op Q : NeTyL -> Ty [ctor Attrs] . Ops) =
  (eq 'transform[Q[makeStratVars(size(NeTyL))], 'N:Nat] =
    'matchrew_s.t._by_['_`[_`]][upTerm(Q), 'makeOpVars[upTerm(NeTyL), 'N:Nat]],
                     'nil.EqCondition,
                     'makeUsingPairs[upTerm(NeTyL),
                        wrapStratList(makeStratVars(size(NeTyL))),
                        '_+_['N:Nat, upTerm(size(NeTyL))], 'N:Nat]
    ] [none] .)
  generateCongOpsDefs(Ops) .

eq generateCongOpsDefs(op Q : nil -> Ty [ctor Attrs] . Ops) =
  (eq 'transform[qid(string(Q) + ".Strategy"), 'N:Nat] =
    'match_s.t._[makeConstant(Q, Ty), 'nil.EqCondition] [none] .)
  generateCongOpsDefs(Ops) .
eq generateCongOpsDefs(Op Ops) = generateCongOpsDefs(Ops) [owise] .
```

The second equation generates the **`matchrew`** constructs described at the beginning of the section, and simpler **`match`** tests are used for constants. Variable names are generated from the index passed as the second argument of `transform`, which is increased in recursive calls to ensure that the index is not used again in a subterm. Just like when declaring them, the same congruence operator may receive multiple `transform` equations for different overloaded data constructors, so they are combined afterwards in a strategy disjunction.

```
eq combineCongOpsDefs(Eqs
     (eq 'transform[T, 'N:Nat] = T1 [none] .)
     (eq 'transform[T, 'N:Nat] = T2 [none] .)) =
  combineCongOpsDefs(Eqs eq 'transform[T, 'N:Nat] = '_|_[T1, T2] [none] .) .
eq combineCongOpsDefs(Eqs) = Eqs [owise] .
```

Generic traversals. Since the strategy language does not provide the means to perform generic traversals of terms, and since we have chosen to translate extended strategies to standard ones, we should implement generic traversals using module-specific strategies. Namely, we can translate the strategy **`gt-all`**(α) to the disjunction of $f(\alpha, \ldots, \alpha)$ for all $f \in \Sigma$, and **`gt-one`**(α) using the disjunction for all f of

$$f(\alpha, \texttt{idle}, \ldots, \texttt{idle}) \;\texttt{or-else}\; \cdots \;\texttt{or-else}\; f(\texttt{idle}, \texttt{idle}, \ldots, \alpha).$$

These still extended strategies are translated to the standard language as explained before. For instance, the strategies in the disjunction to which `gt-all` is translated can be built with the following equations:

```
op generateGTAll : OpDeclSet -> TermList .

eq generateGTAll(none) = empty .
eq generateGTAll(op Q : NeTyL -> Ty [ctor Attrs] . Ops) =
  Q[repeatTerm('S:Strategy, size(NeTyL))], generateGTAll(Ops) .
eq generateGTAll(op Q : nil -> Ty [ctor Attrs] . Ops) =
  makeConstant(Q, 'Strategy), generateGTAll(Ops) .
eq generateGTAll(Op Ops) = generateGTAll(Ops) [owise] .
```

The final shape of the extended metalevel with congruence operators and generic traversals is given by the equation below. Generic traversals are defined directly with equations instead of using `transform`, because they do not explicitly mention any variable indices.

```
eq extendCongOps(M) = fmod append('META-LEVEL, getName(M)) is
  (extending 'SLANG-EXTENSION-STATIC .)
  sorts none .
  none  *** subsorts
  combineCongOps(generateCongOps(getOps(M)))
  (op 'gt-all  : 'Strategy -> 'Strategy [none] .)
  (op 'gt-one  : 'Strategy -> 'Strategy [none] .)
  (op 'gt-some : 'Strategy -> 'Strategy [none] .)
  none  *** membership axioms
  combineCongOpsDefs(generateCongOpsDefs(getOps(M)))
  (eq 'gt-all['S:Strategy] = '_|_[generateGTAll(getOps(M))] [none] .)
  (eq 'gt-one['S:Strategy] = '_|_[generateGTOne(getOps(M))] [none] .)
  (eq 'gt-some['S:Strategy] = '_|_[generateGTSome(getOps(M))] [none] .)
endfm .
```

The `META-LEVEL` module is imported transitively via the module `SLANG-EXTENSION-STATIC`. The complete specification is available in the `congruence.maude` file of the strategy language example collection.

Object-level usage. Writing extended strategies like

```
f(r1[none]{empty}, gt-all(match '0.Zero s.t. nil))
```

at the metalevel and executing them with `metaSrewrite` is possible with what we have introduced so far. However, we want to be able to write expressions like `f(r1, gt-all(match 0))` at the object level, and use them anywhere a standard strategy can be used. This kind of extensions cannot be directly handled by the Core Maude interpreter, but they can be supported through Full Maude or by custom interactive interfaces. The second option has been chosen.

As stated at the beginning of the section, a parser for the augmented strategy language at the object level is required, for which an extensible grammar of the standard strategy language is provided as the `SLANG-GRAMMAR` in Figure 7.2. The developer of the extension should complement it with the productions for the new strategy combinators. Moreover, some module-dependent productions available in the standard strategy language (rule labels for their application, sort membership tests, ...) can be added with a function provided in the skeleton. Thus, we should extend `SLANG-GRAMMAR` as we did with `META-LEVEL`. In addition to the grammar of expressions, a limited grammar of strategy modules is already specified to parse those with extended strategies in their definitions, which are translated to the builtin subset in the transformed module. Finally, a parametric object-oriented module specifies an interactive interface with an adapted `srewrite` command, and where extended modules can be entered. It uses the external objects of Maude 3 for standard input/output communication and the above procedure to parse and execute the strategies. The same procedure is followed by a simple Python script before passing the problem data to the unified model-checking library umaudemc, where the internal and external model checkers are used to verify LTL, CTL*, and μ-calculus properties.

For example, the layered normalization strategy `norm-via-munorm` of Section 7.1 can be specified using a short recursive definition with generic traversals that resume μ-normalization in all arguments of the term. The translation from equations (and from `strat` annotations to `frozen` annotations) in `LAZY-LIST`, which was automatically done by the previous transformation, has been manually done here.

```
smod LAZY-LIST-STRAT is
  protecting LAZY-LIST-RLS .

  strat norm-via-munorm @ LazyList .
  sd norm-via-munorm := one(all) ! ; gt-all(norm-via-munorm) .
endsm
```

The strategy definition is completely generic, despite being parsed in the particular `LAZY-LIST-RLS` module and translated to the standard subset according to it. In fact, the transformed strategy is essentially the same obtained in Section 7.1.

```
        ** Strategy language extensions playground **

SLExt> smod LAZY-LIST-STRAT is … endsm
Module LAZY-LIST-STRAT is now the current module.
SLExt> srew take(3, natsFrom(0)) using norm-via-munorm .
Solution 1:   0:1:2:nil
No more solutions.
```

Another example, where congruence operators are used, is the following module that defines two constants `a` and `b`, a binary function `f`, a rule `swap` that swaps the entries of `f`, and another rule `next` that rewrites `a` to `b`.

```
mod FOO is
  sort Foo .
  ops a b : -> Foo [ctor] .
  op  f   : Foo Foo -> Foo [ctor] .

  vars X Y : Foo .
  rl [swap] : f(X, Y) => f(Y, X) .
  rl [next] : a => b .
endm
```

The extended strategy `f(swap, gt-all(next))` can then be executed:

```
SLExt> select FOO .
Module FOO is now the current module.
SLExt> srew f(f(a,b), f(a,a)) using f(swap, gt-all(next)) .
Solution 1:     f(f(b, a), f(b, b))
No more solutions.
```

Moreover, using the `umaudemc`-based program, temporal properties can be checked on the execution of extended strategies. For instance, checking the formula `False` for the previous strategy provides an arbitrary execution as counterexample. Although we have not used any atomic proposition in this case, the module `FOO` has to be extended as usual by instantiating the `SATISFACTION` elements and including the `STRATEGY-MODEL-CHECKER` module.

```
$ ./slangExtension.py congruenceOpsExt.maude CongOps foo.maude
                       'f(f(a,b), f(a,a))' False 'f(swap, gt-all(next))'
× The property is not satisfied (4 system states, 4 rewrites, 1 Büchi state).
| f(f(a, b), f(a, a))
v  rl f(X, Y) => f(Y, X) [label swap] .
| f(f(b, a), f(a, a))
v  rl a => b [label next] .
| f(f(b, a), f(b, a))
v  rl a => b [label next] .
X f(f(b, a), f(b, b))
```

The files congruenceOpsExt.maude and foo.maude contain the language extension and the example module, respectively, and CongOps is the name of a view that instantiates the different components of the extension skeleton with the concrete extension. These details have been omitted from this exposition, but are explained in the source files.

This procedure can be easily generalized to allow modifications on the module where strategies are applied, or to be parametric on the strategy expression too. For example, inline strategy definitions like **let** *st* $(t_1$, ..., $t_n)$:= β **in** α can be implemented by pushing the definition in the expression to the target module.

7.3 The strategy language semantics as a strategy-controlled system

Strategies are useful to specify semantics of programming languages, as we have seen in previous examples. Moreover, they have been proposed as a general tool to define modular structural operational semantics [BV07] that may easily include ordered rules or negative premises. This example is a straightforward specification with strategies of the Maude strategy language small-step operational semantics presented in Section 3.5, which can be used to model check any strategy-controlled system using the strategy-aware model checker with a fixed strategy. Strategies at the semantics level are in charge of handling the negative case of the conditional and specifying the relations $\to_c$, $\to_s$ and $\twoheadrightarrow$. Obviously, this approach is not recommended to model check strategy-controlled systems in practice, since checking them directly will be much more efficient, but we hope it will be useful to illustrate the usage of strategies to specify semantics without introducing a new language, and to clarify the semantics in Section 3.5 and its relation to model checking. Moreover, the example can be used to experiment with extensions of the strategy language or the model checker, as we will show at the end of the section.

The syntax and semantics of the Maude strategy language depend essentially on the system module being controlled. Hence, the specification of its small-step operational semantics should be parametric on it. Terms, strategies, modules, substitutions, and so on are represented at the metalevel as declared in the predefined META-LEVEL module to simplify the specification and usage of the semantics. The sorts Term of terms, Strategy of strategies, and Module of modules, as well as the different descent functions that allow manipulating them efficiently like metaApply and metaMatch, are described in Sections 2.5.2 and 3.3. Thus, the parameter of the specification can be formalized in the following MODULE theory:

```
fth MODULE is
  protecting META-MODULE .
  op M : -> Module .
endfth
```

As described in Section 3.5, we must specify the *execution state* terms, their rules, and some strategies. Execution states are described as terms of sort ExState using auxiliary sorts like CtxStack for stacks of pending strategies and variable contexts, with the empty-stack symbol eps (ε); and SubtermSoup for the substates of the subterm states.

```
sorts ExState ExStatePart SubtermSoup SolutionSoup CtxStack .
subsort Term < ExStatePart .
subsort SolutionSoup < SubtermSoup .

op _@_ : ExStatePart CtxStack → ExState [ctor] .
op subterm : SubtermSoup Term → ExStatePart [ctor] .
op rewc : Term ExState Substitution Condition StrategyList CtxStack Term
          Context Term → ExStatePart [ctor frozen] .

subsort Strategy < CtxStack .
op ctx : Substitution → CtxStack [ctor] .
op eps : → CtxStack [ctor] .
op __  : CtxStack CtxStack → CtxStack [ctor assoc id: eps] .

op _:_ : Variable ExState → SubtermSoup [ctor] .
op _,_ : SubtermSoup SubtermSoup → SubtermSoup [ctor assoc] .

mb (V : T @ eps) : SolutionSoup .
op _,_ : SolutionSoup SolutionSoup → SolutionSoup [ctor ditto] .
```

The subtype `SolutionSoup` of `SubtermSoup` contains those soups in which all nested states are solutions `T @ eps`. The term projection cterm : $\mathcal{XS} \to T_\Sigma$ is described equationally:

```
op cterm : ExState → Term .
eq cterm(T @ C) = T .
eq cterm(rewc(V, X, Sb, C, SL, Th, R, Ctx, T) @ C) =  T .
eq cterm(subterm(SbS, T) @ C) = applySubs(T, ctermSubs(SbS)) .

op ctermSubs : SubtermSoup → Substitution .
eq ctermSubs(V : X) = V ← cterm(X) .
eq ctermSubs((V : X), SbS) = ctermSubs(V : X) ; ctermSubs(SbS) .
```

where the `applySubs` function applies a substitution to a term, and `ctermSubs` builds the substitution from the variables of the **`matchrew`** to the current subterm being rewritten.

The semantic rules in Section 3.5 are represented almost directly as Maude rules. Their complete relation can be found in the `opsem.maude` file in the example collection, so here we will only show some of them. In general, control rules are labeled with `ctl` and system rules with `sys` so that strategies can distinguish them later. This distinction is extended further to some rules like `ileave` and `ienter` for future application in Section 7.3.2.

```
rl  [ctl]    : T @ idle S ⇒ T @ S .
rl  [ileave] : T @ (A *) S ⇒ T @ S .
rl  [ienter] : T @ (A *) S ⇒ T @ A (A *) S .
crl [ctl]    : T @ (match P s.t. C) S ⇒ T @ S
 if metaMatch(M, P, T, C, 0) :: Substitution .
```

Other rules are defined using auxiliary operators, either predefined like `metaMatch` in the `match` operator rule, or written for the occasion like in the following rules:

```
crl [ctl] : T @ (matchrew P s.t. C by UPS) S
         ⇒ subterm(subtermSoup(UPS, Sb),
                   putInContext(applySubs(P, removeVarsFromSb(Sb, UPS)),
                   Ctx)) @ S
 if {Sb, Ctx} ▷ MPS := metaMatch(M, P, T, C) .
rl [ctl] : subterm(SlS, T) @ S ⇒ applySubs(T, ctermSubs(SlS)) @ S .
```

The first rule initiates a `subterm` state for the `matchrew` and builds all of its components, and the second one concludes the subterm rewriting execution when all of their substates are solutions, since `SlS` is a variable of sort `SolutionSoup`. Since the semantics of rewriting logic itself allows rules to be applied inside subterms, the rules that apply steps inside substates are not needed. The same would be applied to the `rewc` operator for rule rewriting conditions, but both control and system transitions inside its substate should be considered control transitions for the whole `rewc` state, so the `frozen` attribute is added to the operator declaration –which prevents implicit rewrites inside its arguments– and rules are applied explicitly using the following `rewc` rule with its rewriting condition controlled by a strategy, as we will see soon.

```
crl [rewc] : rewc(P, X, Sb, C, SL, CS, RR, Ctx, ST) ⇒
    rewc(P, Y, Sb, C, SL, CS, RR, Ctx, ST) if X ⇒ Y .
```

Another interesting rule is that of strategy calls. It uses the auxiliary function `metaStratDefs` to calculate the matching contexts of the instantiated call term into the definitions of the module. These are returned as a ▷-separated set, so that the rule selects one of them nondeterministically.

```
crl [ctl] : T @ Q[[TL]] S ⇒ T @ CS S
 if CS ▷ CSS := metaStratDefs(M, Q[[reduced(applySubs(TL, vsubs(S)))]]) .
eq ctx(Sb) ctx(Th) = ctx(Sb) .
```

The previous equation implements the tail-recursive call optimization, by removing the lowest of any pair of consecutive contexts in the stack.

Rule applications are handled using an overloaded `metaXapply` function that collects as a set the results of the builtin `metaXapply` descent function. The values in the initial substitution `Sb` are instantiated with the variables of the context and reduced.

```
crl [sys] : T @ Q[Sb]{empty} S ⇒ T' @ S
 if T' ▷ TS := metaXapply(M, T, Q, reduced(applySubs(Sb, vsubs(S)))) .
```

When strategies for rewriting conditions are specified, the state is rewritten to a `rewc` execution state, but we refer the interested reader to the complete specification for the details.

On top of all these rules, strategies are used to specify the $\to_s$, $\to_c$, $\to_{s,c}$, $\twoheadrightarrow$ relations, and the $\twoheadrightarrow^*$ search for solutions that have been extensively used in Sections 3.5 and 4.2. Their definitions are simple:

```
strats →s →c →sc →» opsem @ ExState .

var T  : Term .
var QS : QidSet .

sd →»  := →c * ; →s .
sd →sc := →s | →c .
sd →c  := ctl | ienter | ileave | rewc{→sc}
              | else{not(→sc* ; match T @ eps)} .
sd →s  := sys .
```

The definition of the control transition `→c` includes two other labels in addition to `ctl`. One is the rule `rewc` that applies transitions inside the substate of a `rewc` state, which should be considered control steps no matter whether they are control or system steps in the substate, as explained before. For that reason, the strategy applied to the substate is $\to_{s,c}$. The other label, `else`, refers to the rule for the negative-branch rule of the conditional, defined as

```
crl [else] : T @ (A ? B : G) S ⇒ T @ G S if T @ A vctx(S) ⇒ X [nonexec] .
```

Its rewriting condition is controlled by a strategy that fails if `→sc* ; match T @ eps` succeeds, in other words, if a solution is reachable from `T @ A vctx(S)`, as required by the original rule. Finally, the `opsem` definition

```
sd opsem := test(→c * ; match T @ eps) ? idle : (—» ; opsem) .
```

captures the requirements of the strategy-controlled model described in Definition 3.2: it allows both infinite executions of —» transitions, and finite ones ending in states where a solution can be reached by control transitions. Ensuring that the —» transition is seen as an atomic step, for what the opaque strategy feature described in Section 4.3.1 can be used, the system controlled by `opsem` from the initial state $\bar{t}$ `@` $\bar{\alpha}$ is equivalent to t controlled by α modulo the `cterm` projection. The `matchrew` combinator is executed without the partial order reduction feature explained in Section 4.3.1, but a biased version can be programmed with strategies using `matchrew`, insisting in the reflective nature of this example.

The last requirement for model checking is defining atomic propositions. Since states and strategies are represented at the metalevel, atomic propositions are also represented as metaterms.

```
mod NOP-PREDS{X :: MODULE} is
  protecting NOP-RULES{X} .
  including SATISFACTION .

  subsort ExState < State .
  op prop : Term → Prop [ctor] .

  var XS : ExState .
  var P  : Term .
  eq XS ⊨ prop(P) = reduced('_⊨_[cterm(XS), P]) == 'true.Bool .
endm
```

The predicate term is wrapped in a `prop` symbol, whose satisfaction is defined using the predefined `metaReduce` function that evaluates `cterm(`q`)` $\models p$ in the base module, where q and p are the terms metarepresented by `XS` and `P` respectively. The `NOP-PREDS` module is parameterized by the `MODULE` theory, which determines the underlying module.

For example, we can instantiate the semantics with the dining philosophers specification of Section 6.2. The formal constant `M` in the `MODULE` theory is mapped to the metarepresentation of the `DINNER-MCS` module obtained with the builtin `upModule` operator.

```
view Philosophers from MODULE to META-LEVEL is
  op M to term upModule('DINNER-MCS, true) .
endv
```

To model check the formula $\Box\Diamond \bigvee_{k=1}^{4}$ `eats(`k`)` from the initial term `initial` using the `parity` strategy, we only have to model check the execution state $\bar{\iota}$ `@` $\bar{\alpha}$ combining the metarepresentations of `initial` and `parity` against the property whose atomic propositions have been replaced by their metarepresentations inside the `prop` symbol. The semantics is executed under the control of `opsem` with —» as opaque strategy to respect the transitions of the original model.

```
Maude> red modelCheck(reduced('initial.Table) @ 'parity[[empty]],
             [] <> (prop('eats['0.Zero]) \/ ...
                    \/ prop('eats['s_^4['0.Zero]])), 'opsem, '—») .
rewrites: 454394
result Bool: true
```

In case the property did not hold, we would obtain a counterexample whose states are execution states of the semantics. In order to show more readable counterexamples and check branching-time properties too, a small `umaudemc`-based program is available together with the Maude specification in the example collection.

Model checking strategy-controlled systems using this executable semantics is overkill. Still, introducing changes or new features to this specification is fairly quick and simple, and it can be used to experiment with variations of the strategy language, which can be directly executed and model checked. The following sections show two such applications.

7.3.1 Extending the language with new operators

In Section 3.8.1, we have seen that the strategy language introduced by Meseguer along with the Temporal Logic of Rewriting [Mes07] includes two operators, $\alpha \wedge \beta$ and $\alpha \, \mathbf{U} \, \beta$, that are not available in the official Maude strategy language. Extending the executable small-step operational semantics, we will implement the first operator. Remember that $\alpha \wedge \beta$ denotes all rewriting paths that are allowed by both α and β, including those in which a strategy concludes before the other. The approach of Section 7.2 cannot be used to implement this operator, unless the contents of α and β are dissected, because it is not expressible in the current strategy language.

Extending the semantics is as simple as extending the modules where it is defined with new operators, strategies, and rules. In the following parameterized module `NOP-AND-RULES`, we declare the binary strategy constructor `_/\_` in the `Strategy` sort, and the execution state `sync` that will be used to implement it.

```
mod NOP-AND-RULES{X :: MODULE} is
  extending NOP-RULES{X} .

  op _/\_  : Strategy Strategy -> Strategy [ctor assoc] .
  op sync : ExState ExState -> ExStatePart [ctor comm frozen (1 2)] .

  vars X Y X' Y' : ExState .
  var  P         : ExStatePart .
  vars C S       : CtxStack .
  vars A B       : Strategy .
  var  T         : Term .

  eq cterm(sync(X, Y) @ C) = cterm(X) .
```

The commutative `sync` operator consists of two execution states that advance simultaneously to execute the strategies given to the $\wedge$ combinator. We define its `cterm` projection as the projection of any one of the states, since both are rewritten in parallel to the same terms by the following rules.

```
  rl [ctl] : T @ (A /\ B) S => sync(T @ A vctx(S), T @ B vctx(S)) @ S .

  crl [sync] : sync(X, Y) => sync(X', Y) if X => X' .
  crl [sync] : sync(X, Y) => sync(X', Y') if X => X' /\ Y => Y'
   /\ reduced(cterm(X')) = reduced(cterm(Y')) .

  rl [ctl] : sync(T @ eps, P @ C) @ S => P @ (C S) .
endm
```

The first rule creates the `sync` state when the intersection strategy is on top of the stack, and the last one destroys it when any of the synchronized substates has reached a solution. The rules in the middle are in charge of executing control and system steps on the substates with the help of strategies. Control steps can be applied discretionally, but system steps must be executed simultaneously on both substates and their resulting terms must coincide to guarantee that they are executing the same rewriting paths. The rewriting fragments of these rules are guided by strategies in the definition of the →c and →s strategies of the semantics.

```
smod NOP-AND-SEMANTICS{X :: MODULE} is
  protecting NOP-AND-RULES{X} .
  extending NOP-SEMANTICS{X} .

  sd ->c := sync{->c} .
  sd ->s := sync{->s, ->s} .
endsm
```

According to the semantics of the strategy calls, these alternative definitions for the arrow strategies will be executed as if they were inserted with a disjunction operator in the original ones. This clean and short specification is all we need to execute and model check with the intersection operator.

After instantiating the extended semantics with the `Philosophers` view, the following command rewrites the intersection of the strategy `left ; right` with a strategy that sequentially selects the philosophers one and two with two **matchrew**s and applies any of the two rules to them.

```
Maude> srew reduced('initial.Table) @ ('left[none]{empty} ; 'right[none]{empty}
  /\ (amatchrew 'L:List s.t. '__['B1:Being, '_|_|_['O1:Obj, 's_['0.Zero],
        'O2:Obj], 'B2:Being] := 'L:List by 'L:List using all) ;
     (amatchrew 'L:List s.t. '__['B1:Being, '_|_|_['O1:Obj, 's_^2['0.Zero],
        'O2:Obj], 'B2:Being] := 'L:List by 'L:List using all))
       using opsem ; ->c ! .

Solution 1
rewrites: 4034
result ExState: '<_>['__['_|_|_['o.Obj,'0.Zero,'o.Obj],
                         '_|_|_['ψ.Obj,'s_['0.Zero],'o.Obj],'ψ.Obj,
                         '_|_|_['o.Obj,'s_^2['0.Zero],'ψ.Obj],
                         '_|_|_['o.Obj,'s_^3['0.Zero],'o.Obj],'ψ.Obj,
                         '_|_|_['o.Obj,'s_^4['0.Zero],'o.Obj],'ψ.Obj]] @ eps

No more solutions.
rewrites: 4081
```

The result is that the philosopher one takes the left fork, and then the philosopher two takes the right fork, satisfying both restrictions.

There is an alternative specification of this operator in the example collection for an arbitrary number of synchronized strategies instead of two. This is used to implement the other missing operator $\alpha \mathbf{U} \beta$, although its precise semantics would require the Kleene-star variation of Section 3.6.1 to ensure that β is eventually executed.

7.3.2 Tracing iterations for the Kleene-star model checker

When a rewriting system is controlled by the semantics of the strategy language whose iteration is a Kleene star, the restricted model cannot be represented in general as a plain Kripke structure. The model checker prototype presented in Section 4.5 proceeds by generating a Büchi automaton for the model and calculating its intersection with the property automaton, or by transforming the formula to include the restrictions on the finite iterations. In either case, we need to know when iterations are started and abandoned within the rewrite graph, for what they use the extension in this section, which keeps track of these circumstances and annotates the states with the required information.

In the following module `NOP-KLEENE-SEMANTICS` extending `NOP-RULES`, execution states are wrapped into a pair whose second component is a set of tags of sort `ActionTag` that describe whether the state is the result of a transition that `enter`s or `leave`s an iteration.

```
smod NOP-KLEENE-SEMANTICS{X :: MODULE} is
  extending NOP-RULES{X} .

  sorts WraptState PartialContext ActionTag ActionTags .
  subsort CtxStack < PartialContext .

  op wrap : ExState ActionTags → WraptState [ctor frozen(2)] .
  op enter : PartialContext → ActionTag [ctor format (y o)] .
  op leave : PartialContext → ActionTag [ctor format (y o)] .
```

Moreover, the iteration is identified by its partial context defined in Section 4.5, which includes the concatenation of all context stacks and `matchrew` variables from the subterm where the iteration is directly applied to the outermost context. The `PartialContext` sort is defined as a supersort of `CtxStack` with a new constructor `sub` for inserting variables into the stacks.

```
op sub : Variable → PartialContext [ctor] .
op __  : PartialContext CtxStack → PartialContext [ditto] .
op __  : CtxStack PartialContext → PartialContext [ditto] .
```

Executing this semantics is as usual a succession of ⟶» steps until a solution is found or indefinitely. However, the ⟶» step of the semantics is not defined by `→c * ; →s`, but via a recursive exploration of the term. Before that, the tags of the initial state of the transition are discarded, since they refer to the transition that originates the state.

```
strats ⟶» ⟶»r →cw opsem @ WraptState .
sd ⟶» := discardTags ; ⟶»r .
rl [discardTags] : wrap(XS, As) ⇒ wrap(XS, none) [nonexec] .
```

Then, the recursive strategy ⟶»`r` is called. If the state where it is applied is not a `subterm` state, it executes either a system transition or a control transition →`cw` before trying ⟶»`r` again. The new control step →`cw` executes a single control rule as ever, except that iteration rules are applied explicitly with a `matchrew` that adds the corresponding tag to the wrapped state with the rule `addTag`. Otherwise, if ⟶»`r` is applied to a `subterm` state, the recursive strategy is tried on any of its subterms through the `recurseSubterm` rule. Apart from rewriting the substate, this rule gets the tags produced for it and extends them with the context of the outer one. The function `extendActionTags` simply appends the partial contexts of the tags with the given argument. Thanks to this recursive traversal, iteration applications are tagged with the appropriate partial context.

```
sd ⟶»r := match wrap(subterm(Sbs, T) @ C, As)
  ? recurseSubterm{⟶»r}
  : (→s | →cw ; ⟶»r) .

sd →cw := (ctl | else{not(opsem-sc)} | rewc{→sc})
  | (matchrew wrap(XS, As) s.t. T @ C := XS
      by XS using ienter, As using addTag[A ← enter(C)])
  | (matchrew wrap(XS, As) s.t. T @ C := XS
      by XS using ileave, As using addTag[A ← leave(C)])
.

crl [recurseSubterm] : wrap(subterm(((X : XS1), Sbs), T) @ C, As)
                    ⇒ wrap(subterm(((X : XS2), Sbs), T) @ C,
                           As extendActionTags(As', sub(X) C))
 if wrap(XS1, none) ⇒ wrap(XS2, As') [nonexec] .

rl [addTag] : As ⇒ A As [nonexec] .
```

The strategies →`s`, `opsem-sc` and →`sc` mentioned above are defined as in the original specification. Finally, the only change in `opsem` is that the `wrap` constructor is taken into account.

```
sd opsem := test(matchrew wrap(XS, As) by XS using (→c * ; match T @ eps))
          | (⟶» ; opsem) .
```

Like in the NOP-PREDS module of the base semantics, atomic propositions are defined in a module NOP-KLEENE-PREDS to check properties of the underlying system through the wrapped states and to check the iteration tags. The automata-based model checker for the Kleene-star semantics checks the tags directly to set up the Streett conditions of the system automaton, while the modified formulae use the atomic propositions defined in this module. Slight modifications allow checking also when a strategy has been called and registering the labels of rule rewrites, in order to implement the improvements suggested in Section 4.5 and to provide as much information as the builtin model.

7.4 Multistrategies

The paradigm of strategy-controlled systems proposed in Maude is the combination of a rewrite system and a strategy expression that controls it as a whole. However, many systems are better specified compositionally. A typical example are object- or agent-oriented systems, in which each object or agent would follow its own strategy. Likewise, describing the interaction of players in games with a single sequential control flow is cumbersome. Hence, we propose the following model transformation to facilitate this specification problem. Instead of a single strategy expression α, the system control will be specified by a *multistrategy*: an undetermined number of strategies $\alpha_1, \ldots, \alpha_n$ and a global strategy γ that describes how they are combined. Two builtin γ are provided: a concurrent one, in which the next strategy to take a step can be any of them, and a turn-based one, in which strategies are executed in a fixed order. A fundamental question is the amount of atomic work done by a strategy α_i when it is given control. In other words, the granularity of their interleaving in the global execution. Imitating the small-step operational semantics of the strategy language, a single rule application is a reasonable atomic step, but a few more strategies should be executed atomically like matchrews with a non-trivial pattern and conditions in the conditional operator, since they assume a particular structure or invariant of the term that may not be preserved if another strategy *thread* modifies the term during the while.

Multistrategies are implemented using strategies at the metalevel and an augmented execution environment. Essentially, to evaluate the strategies α_1, ..., α_n on the subject term t, they are transformed into the term { $\overline{t}$ :: < 1 % $\overline{\alpha}_1$ > ⋯ < n % $\overline{\alpha}_n$ >, $\overline{M}$ } that includes the metarepresentation of the subject term t, of the strategies α_i, and of the module M where they are evaluated. The evolution of this execution context is defined by some rules, which modify the strategy representations and execute them according to their semantics, governed by the global strategy γ. Coherently with the terminology for the strategy language semantics, the rules that do not alter the subject term (but choose alternatives, expand iterations...) are called *control rules*, and those modifying the term with rules of the underlying system are called *system rules*. With control(N) and system(N) being the disjunction of all control and system rules applied to the thread N, global control strategies, like turn(N, M) for executing M strategies in turns starting from the N$^{\text{th}}$ one and freec to execute them concurrently, can be specified as follows:

```
vars N M K : Nat . var T : Term . var  : Strategy . var Mod : Module .

sd ⟹›(N) := control(N) * ; system(N) .
sd turns(N, M) := ⟹›(N) ? turns(s(N) rem M, M) : idle .
sd freec := (matchrew C s.t. { T :: < N % S > TS, Mod } := C
               by C using ⟹›(N)) ? freec : idle .
```

The ⟹› strategy specifies the atomic step of a strategy thread execution as explained before. The definitions of the γ strategies turns and freec apply this atomic step ⟹› with indices that are respectively increased cyclically or selected nondeterministically by matching. Notice that the turns strategy stops when the current strategy is unable to continue, while freec halts when all threads are stuck. Custom global strategies can be easily defined for other general or specific purposes. For example, the `freec` strategy can be bound on the number of steps:

```
sd freec(0) := idle .
sd freec(s(K)) := (matchrew C s.t. { T :: < N % S > TS, Mod }
                 := C by C using ⟹»(N)) ? freec(K) : idle .
```

Further details on the transformation are discussed in Section 7.4.2, and the complete commented Maude code is available in the example collection.

Auxiliary operations and an interactive environment have been prepared to easily execute multistrategies at the object level, and to obtain meaningful counterexample traces when model checking these systems. The interactive environment is similar to that used for the language extensions in Section 7.2, with a command for rewriting with multistrategies `srewrite` t `using` α_1, ..., α_n `by` γ where γ can be the words `turns` or `concurrent` for the predefined strategies, or `custom` followed with an arbitrary expression. Another command `check` φ `from` t `using` α_1, ..., α_n `by` γ checks the LTL property φ on the given multistrategic model. Branching-time properties can also be checked using an external `umaudemc`-based command line tool.

Let us illustrate the execution of multistrategies with the simple `LLIST` example below.

```
smod LLIST is
  sorts Letter List .
  subsort Letter < List .
  ops a b c d e : → Letter [ctor] .
  op nil : → List [ctor] .
  op __ : List List → List [ctor assoc comm id: nil] .

  var L : Letter . var LS : List .
  rl [put] : LS   ⇒ LS L [nonexec] .

  strat seq : List @ List .
  sd seq(nil)  := idle .
  sd seq(L LS) := top(put[L ← L]) ; seq(LS) .
endsm
```

The module specifies a list of letters (`a`, `b`, `c`, ...) that can be appended with a `put` rule, and a strategy `seq` that does so with a list of them in order. After loading that module and the interactive interface in `multistrat-iface.maude`, we can execute multiple `seq` calls by turns or concurrently:

```
    ** Multistrategies playground **

MStrat> select LLIST .
MStrat> srew nil using seq(a b), seq(c d) by turns .
Solution 1:   a c b d
No more solutions.

MStrat> srew nil using seq(a b), seq(c d) by concurrent .
Solution 1:   a b c d
…
Solution 6:   c d a b
No more solutions.
```

Letters appear in a zipper sequence when the strategies are executed by turns, and in all possible interleavings when they are executed concurrently.

More interesting examples have been specified using multistrategies, including the Lamport's bakery algorithm and the tic-tac-toe game, where relevant properties are model checked using different combinations of process or player strategies. This latter example is studied in the following section.

7.4.1 Multistrategies playing games: the tic-tac-toe

Tic-tac-toe or *noughts and crosses* is a popular game in which two players, circles and crosses, take turns putting their symbols in a 3x3 grid to complete a vertical, horizontal, or diagonal sequence of cells. The first player to achieve the goal is the winner. Tic-tac-toe is a solved game that always ends in a draw if no player makes a mistake. In this section, we will specify a flawless strategy for a player, and use the model checker to prove that it actually is. But before that, the representation of the board and the rules should be specified.

```
fmod TICTACTOE is
  protecting NAT .
  protecting EXT-BOOL .

  sorts Position Player Grid .

  ops O X - : → Player [ctor] .

  op [_,_,_] : Nat Nat Player → Grid [ctor] .
  op empty   : → Grid [ctor] .
  op __      : Grid Grid → Grid [ctor assoc comm id: empty] .
```

The `Grid` sort's elements are sets of triples that map a coordinate on the game board to the player that occupies that position, `O` for circles, `X` for crosses, and `-` to mean an empty position. In order to decide whether a game is finished, some predicates are defined to detect complete rows in every possible direction.

```
  ops hasHRow hasVRow hasDRow : Player Grid → Bool .

  vars I1 I2 I3 J : Nat .  var G : Grid .  var P : Player

  eq hasHRow(P, [I1, J, P] [I2, J, P] [I3, J, P] G) = true .
  eq hasHRow(P, G) = false [owise] .
```

As a commutative and associative operator, the grid matches the definition pattern if the entire row `J` belongs to player `P` for some `J`. Predicates for the other directions are defined similarly. Using them, winning is defined by the following disjunction:

```
  op hasWon : Player Grid → Bool .
  eq hasWon(P, G) = hasHRow(P, G) or-else hasVRow(P, G) or-else hasDRow(P, G) .
endm
```

where `or-else` is a short-circuit version of the logical disjunction provided by the predefined module `EXT-BOOL`. The game module also defines a constant `initial` for the initial grid of empty positions [k, l, -] for all $1 \leq k, l \leq 3$.

In the system module `TICTACTOE-RULES`, the player movements are represented by two rules `putO` and `putX` that simply place its symbol on an empty position.

```
mod TICTACTOE-RULES is
  protecting TICTACTOE .

  vars I J : Nat .

  rl [putO] : [I, J, -] ⇒ [I, J, O] .
  rl [putX] : [I, J, -] ⇒ [I, J, X] .
```

```
endm
```

In these terms, we can define strategies for playing the game in a strategy module TICTACTOE-STRAT. The simplest and most unconscious strategy is the free application of the put rules, but stopping when the game is over. This is what the following random strategies do:

```
smod TICTACTOE-STRAT is
  protecting TICTACTOE-RULES .

  strats randomO randomX @ Grid .

  vars G R : Grid . vars I1 I2 I3 I J : Nat . var P : Player .

  sd randomO := (match G s.t. not hasWon(X, G) ; putO) ? randomO : idle .
  sd randomX := (match G s.t. not hasWon(O, G) ; putX) ? randomX : idle .
```

Note that random does not mean that the positions are chosen randomly, but that any of them can be chosen, and so the strategy search commands will explore all possible selections. Moreover, randomX and randomO are not mutually recursive, since each represents the strategy of a different player and turns are handled by the multistrategies framework. These strategies can be improved with certain easy intuitions about more clever movements. For example, if the current player already has two positions in a row and the third one is empty, the symbol should be put there to win immediately. Otherwise, if that situation occurs for the opponent, the active player should occupy the empty position to prevent the other player from winning in its next turn. This is specified by the following betterO strategy for the player O (similarly for the X player).

```
  strats betterO betterX @ Grid .

  sd betterO := (match G s.t. not hasWon(X, G) ;
    ((matchrew G s.t. [I, J, -] R := winningPos(O, G)
        by G using putO[I <- I, J <- J])
    or-else
    (matchrew G s.t. [I, J, -] R := winningPos(X, G)
       by G using putO[I <- I, J <- J])
    or-else
    putO)) ? betterO : idle .
```

The winningPos(P, G) function returns a set of sort Grid with all the positions where player P can complete a row. These are calculated equationally, by pattern matching again.

```
  ops winningPos winningHPos winningVPos winningD1Pos
      winningD2Pos : Player Grid -> Grid .

  eq winningPos(P, G) = winningHPos(P, G) winningVPos(P, G)
                        winningD1Pos(P, G) winningD2Pos(P, G) .
  eq winningHPos(P, [I1, J, P] [I2, J, P] [I3, J, -] G) =
       [I3, J, -] winningHPos(P, G) .
  eq winningHPos(P, G) = empty [owise] .
```

Functions to find vertical and diagonal rows are defined similarly. At this point, we can then ask ourselves whether this strategy is *perfect*, i.e. whether it always leads to the best possible outcome: not losing the game no matter how the other player behaves. Using the integrated model checker, we can formally verify it. Making X play with better and O play with random, i.e. trying all possible moves for O, the property $\square \neg$ *Owins* tells us whether better is optimal.

```
MStrat> select TICTACTOE-CHECK .
```

```
Module TICTACTOE-CHECK is now the current module.
MStrat> check [] ~ Owins from initial using betterX, randomO by turns .
| initial
v 0 does putX
| [1, 1, -] ............................................................ [3, 2, -] [3, 3, X]
v 1 does putO
| [1, 1, -] ................................................ [3, 1, -] [3, 2, O] [3, 3, X]
v 0 does putX
| [1, 1, -] .................................... [2, 3, -] [3, 1, X] ........... [3, 3, X]
v 1 does putO
| [1, 1, -] ... [1, 3, -] .................. [2, 3, O] ......................... [3, 3, X]
v 0 does putX
| [1, 1, -] ... [1, 3, X] .......... [2, 2, -] ................................. [3, 3, X]
v 1 does putO
| [1, 1, -] ............... [2, 1, -] [2, 2, O] ................................ [3, 3, X]
v 0 does putX
| [1, 1, -] [1, 2, -] .. [2, 1, X] ............................................. [3, 3, X]
v 1 does putO
X [1, 1, -] [1, 2, O] ............................................................ [3, 3, X]
```

And it is not, since the counterexample (where we have removed the positions that have not changed) shows an execution in which the circles win even if the crosses play the `better` strategy. In fact, the only situation where the intelligence of the strategy is actually used is the last move for `X`, when `O` has two winning positions in the middle vertical and horizontal row that `X` cannot block at the same time.

Hence, `better` is not a perfect strategy, and further precautions should be taken not to make mistakes. In Table 1 of [CS93], we can find the script of a perfect strategy for playing tic-tac-toe, which is similar to the algorithm used by the Newell and Simon's tic-tac-toe program in 1972. This script includes various rules to play, which we will call *actions* not to confuse them with rewrite rules, that are not necessarily exclusive. For the strategy to be effective, these actions must be executed in order, applying at each step the first possible one. The first two are those included in `better`, (1) *Win* that completes the row where there are already two positions of the current player, and (2) *Block* that prevents the opponent from doing so in the next turn.

```
strats perfectO perfectX          @ Grid .
strat  perfect-step      : Player @ Grid .

sd perfectO := (match G s.t. not hasWon(X, G) ;
                perfect-step(O)) ? perfectO : idle .
sd perfectX := (match G s.t. not hasWon(O, G) ;
                perfect-step(X)) ? perfectX : idle .

sd perfect-step(P) :=
  *** Win
  (matchrew G s.t. [I, J, -] R := winningPos(P, G) by G using put(P, I, J))
  or-else
  *** Block
  (matchrew G s.t. [I, J, -] R := winningPos(opponent(P), G)
     by G using put(P, I, J))
  or-else
  *** Fork
  (put(P) ; hasFork(P))
  or-else
  *** Blocking an opponent's fork
```

```
    (test(put(opponent(P)) ; hasFork(opponent(P))) ; put(P) ;
      *** The opponent cannot fork
      (not(put(opponent(P)) ; hasFork(opponent(P)))
      *** The opponent is forced to block rather than fork
      | (matchrew G s.t. [I, J, -] R := winningPos(P, G)
           by G using not(put(opponent(P), I, J) ; hasFork(opponent(P)))))
    )
    or-else
    *** Center
    put(P, 2, 2)
    or-else
    *** Opposite corner
    ((matchrew [I, I, Q] G s.t. I =/= 2 /\ Q = opponent(P)
        by G using put(P, sd(4, I), sd(4, I)))
    | (matchrew [I, J, Q] G s.t. I =/= 2 /\ J =/= 2
        /\ Q = opponent(P) by G using put(P, J, I)))
    or-else
    *** Empty corner
    (put(P, 1, 1) | put(P, 3, 3) | put(P, 1, 3) | put(P, 3, 1))
    or-else
    *** Empty side
    (match G s.t. P == O ? (putO[I <- 2] | putO[J <- 2])
                         : (putX[I <- 2] | putX[J <- 2]))
```

The `or-else` combinator guarantees that actions are applied in order. Not to define the same strategy twice for each player, as we did with the previous shorter strategies, the `perfect-step` strategy takes the player as an argument, and uses the new `put` strategy and `opponent` function.

```
  strat put : Player Nat Nat @ Grid .
  sd put(X, I, J) := putX[I <- I, J <- J] .
  sd put(O, I, J) := putO[I <- I, J <- J] .
  strat put : Player @ Grid .
  sd put(X) := putX .
  sd put(O) := putO .

  op opponent : Player ~> Player .
  eq opponent(X) = O .
  eq opponent(O) = X .
```

After the first two actions, the strategy tries (3) *Fork* to obtain two winning positions for the next turn, so that winning is guaranteed unless the other player completes a row immediately. Instead of calculating these positions equationally, our strategy puts a symbol randomly and then checks whether there is a fork with the following strategy:

```
  strat hasFork : Player @ Grid .
  sd hasFork(P) := match G s.t. size(winningPos(P, G)) >= 2 .
```

The next actions are (4) *Block fork* that prevents the opponent's fork, (5) *Center* that occupies the center position, (6) *Opposite corner* that fills the diagonally-opposite corner of an opponent's position, (7) *Empty corner* that puts the symbol in any corner, and finally (8) *Empty side* that uses a side instead. Note that the *Empty side* action is the only remaining possibility when the previous actions have been discarded, so we could have equivalently written `put(P)` instead of the specific strategy used in the `perfect-step` definition.

Using the `check` command and the same formula again, we discover that `perfect` is actually perfect no matter which player starts.

```
 MStrat> check [] ~ Owins from initial using perfectX, randomO by turns .
```

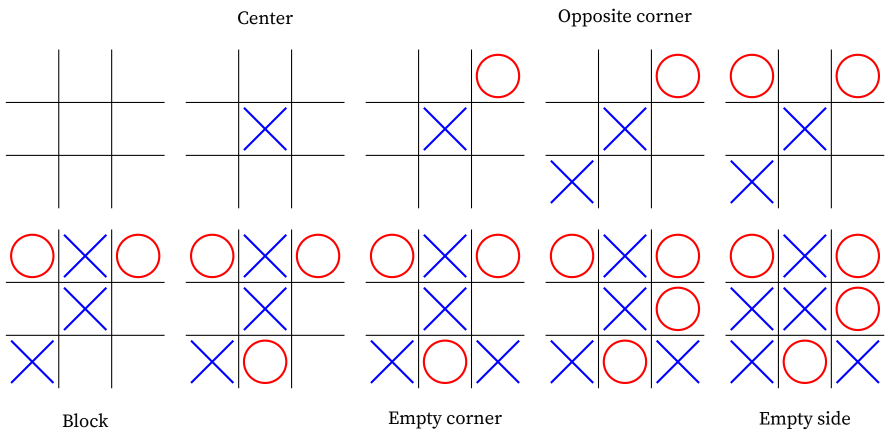


Figure 7.3: Game where `perfectX` does not win against `random0`.

```
The property is satisfied.
MStrat> check [] ~ Owins from initial using random0, perfectX by turns .
The property is satisfied.
```

However, the strategy does not ensure that `X` eventually wins. The game may end in a draw, as attested by the following counterexample, which has been drawn in Figure 7.3.

```
MStrat> check <> Xwins from initial using perfectX, random0 by turns .
| initial
v 0 does perfect-step
| [1, 1, -] ............................ [2, 2, X] ............................ [3, 3, -]
v 1 does put0
| [1, 1, -] ........................................................................ [3, 3, 0]
v 0 does perfect-step
| [1, 1, X] .............................................. [3, 1, -] ........... [3, 3, 0]
v 1 does put0
| [1, 1, X] .............................................. [3, 1, 0] [3, 2, -] [3, 3, 0]
v 0 does perfect-step
| [1, 1, X] [1, 2, -] ................................................ [3, 2, X] [3, 3, 0]
v 1 does put0
| [1, 1, X] [1, 2, 0] [1, 3, -] .............................................. [3, 3, 0]
v 0 does perfect-step
| [1, 1, X] ........... [1, 3, X] ................. [2, 3, -] ................. [3, 3, 0]
v 1 does put0
| [1, 1, X] ...................... [2, 1, -] ..... [2, 3, 0] .................. [3, 3, 0]
v 0 does perfect-step
X [1, 1, X] ........................ [2, 1, X] .................................. [3, 3, 0]
```

In particular, both players can play the `perfect` strategy, and then no one wins.

```
MStrat> check [] (~ Owins /\ ~ Xwins) from initial
        using perfectX, perfect0 by turns .
The property is satisfied.
```

Finally, we wonder whether the perfect strategy we have adopted from [CS93] is concise or it can be simplified. Repeating the `check` commands with variations of the strategy, we can see that the last three actions, *Opposite corner*, *Empty corner*, and *Empty side*, can be replaced by the unrestricted `put(P)`, and so these distinctions are superfluous. However, this simplification cannot be extended to the *Center* action without losing perfection. Other combinations of rules can be safely removed too.

Not only the `check` command is useful, but the `srewrite` command allows obtaining, for instance, all final configurations of the game using certain strategies.

```
MStrat> srew initial using perfectX, randomO by turns .
Solution 1:     [1, 1, -][1, 2, -][1, 3, X]
                [2, 1, -][2, 2, X][2, 3, -]
                [3, 1, X][3, 2, O][3, 3, O]
...
Solution 134:   [1, 1, O][1, 2, X][1, 3, O]
                [2, 1, O][2, 2, X][2, 3, X]
                [3, 1, X][3, 2, O][3, 3, X]
No more solutions
```

7.4.2 A deeper look into the implementation

Unlike the previous metalevel transformations, multistrategies are not handled by an equational static manipulation of the original module and its strategies. Instead, the term to be rewritten and the multiple strategies that act on it are operated at the metalevel during their execution. The execution context for the multistrategies is not strictly parametric on the subject system, but contains it as data while being rewritten in the system module `MULTISTRAT`. The already-seen context $\{ \bar{t} :: <1 \% \bar{\alpha}_1 > \cdots < n \% \bar{\alpha}_n >, \overline{M} \}$ is specified as:

```
sorts   MSContext MSThread MSThreadSet .
subsort MSThread < MSThreadSet .

op <_%_> : Nat Strategy -> MSThread [ctor] .
op none  : -> MSThreadSet [ctor] .
op __    : MSThreadSet MSThreadSet -> MSThreadSet
             [ctor assoc comm id: none] .
op {_::_,_} : Term MSThreadSet Module -> MSContext [ctor] .
```

The key fact that lets us follow the execution of the multiple strategies on the subject term is that contexts are univocally associated to terms, and their transitions to their transformations, as we will see.

```
op getTerm : MSContext -> Term .
eq getTerm({ T :: TS, M }) = T .
```

The multiple strategy threads are run on these contexts by several rules that reimplement at some extent the execution of strategies. As said before, some rules only manipulate and decompose strategies, while others may modify the term being rewritten. For example, the following are control transitions for the iteration and disjunction combinators:

```
rl [ms-reduct] : < N % S * ; Ss > => < N % Ss > .
rl [ms-reduct] : < N % S * ; Ss > => < N % S ; S * ; Ss > .

crl [ms-choose] : < N % (S1 | S2) ; Ss > => < N % S1 ; Ss >
 if S1 =/= fail /\ S2 =/= fail .
```

System transitions are performed by the following rule that executes the strategy `S`:

```
crl [ms-run] : { T  :: < N % S ; Ss > TS, M }
            => { T' :: < N % Ss > TS, M }
  if S =/= idle
  /\ atomicStrategy(S)
  /\ T' ; Ts := allSuccs(M, T, S) .
```

Rule applications are not the only expressions considered `atomicStrategy`, but also `matchrew` strategies with multiple subterms, as we have said, since they assume a fixed structure of the term along all its execution, which may otherwise be broken by another thread acting on the term. The successors of atomic strategies are calculated using the builtin Maude engine via `metaSrewrite`. However, to nondeterministically select one of the possible successors, we collect all of them in a set with the `allSuccs` function, and let the rule be instantiated with each by matching the `T' ; Ts` pattern on that set of terms. Another atomic action is the evaluation of conditions in conditional operators by the following rule:

```
crl [ms-cond] : { T  :: < N % (S1 ? S2 : S3) ; Ss > TS, M }
             => { T' :: < N % S2 ; Ss > TS, M }
 if T' ; Ts := allSuccs(M, T, S1) .
```

Finally, all these rules are gathered in the strategies `control` and `system` that were already mentioned in the overview at the beginning of this section.

```
strats control system : Nat @ MSContext .

sd system(N) := ms-cond[N <- N] or-else ms-run[N <- N] .
sd control(N) := ms-reduct[N <- N] | … | ms-def[N <- N] .
```

The Maude-based interactive interface and the command-line program to verify branching-time properties are programmed similarly to that for the strategy language extensions seen in Section 7.2.

Unlike the previous examples in Sections 7.1 and 7.2, the reflective implementation of multistrategies just explained operates with the metarepresentation of the strategies at *run time*, instead of producing or *compiling* a new module that is then executed by Maude at the object level. Hence, a sometimes noticeable performance penalty is to be expected when executing and model checking with multistrategies. For example, in the Lamport's bakery algorithm mentioned before and available in the example collection, generating the entire state space requires 50 seconds with multistrategies but only 200 ms using a less natural alternative strategy that is also included in this specification. However, a more efficient and complex transformation of the first style or a direct implementation could be developed if multistrategies need to scale for more complex applications.

Chapter 8

Membrane systems

Membrane computing is a parallel and distributed computational model inspired on biological cells. Each cell's behavior is governed by a kind of rewrite rules that are applied according to specific control mechanisms where priorities and some environmental factors are involved. The whole system evolves by the parallel evolution steps of its cells followed by the interchange of messages and possible transformations on the structure. Their control mechanisms make membrane systems a paradigmatic example of strategy-controlled systems. They have been represented in rewriting logic using different approaches [ACL07, ACL05], and even using a primitive version of the Maude strategy language [AL09]. The strategy-controlled representation of membrane systems described in this chapter is based on these works, although strategies are used differently and several extensions are considered. Moreover, the strategy-aware model checker is used to check any supported temporal property on membrane computations, respecting thanks to the opaque strategies in Section 4.3.1 their genuine steps, which are not rule rewrites but aggregations of parallel such rewrites. The general problem of the representation of parallel rewriting can be addressed with the same approach, where some strategies collect all steps that can take place simultaneously, and their complete execution is seen as the parallel composed step.

This chapter is based on the journal article [RMP+22b], which was first presented in the workshop WPTE 2020 [RMP+20b]. The interactive environment for simulating and model checking membrane systems is available within the example collection.

8.1 Membrane systems

Membrane computing [Pău02] is a biologically-inspired computational model where cells are parallel and distributed processing units that communicate by passing objects through their membranes like chemicals traverse those of biological cells. A *membrane system* or *P system* is a collection of cells or membranes populated by a multiset of other nested cells, *objects* playing the role of chemicals, and *evolution rules* describing their reactions and communication. All of them are assumed to be contained inside a single topmost *skin membrane*. Objects are usually opaque identifiers represented by letters, and evolution rules $u \to v$ consist of a multiset u of objects and a multiset v of *targets* of the form (w, t) where w is a multiset of objects and t determines whether these must stay in the membrane (*here*), or be transferred to the enclosing one (*out*) or to a nested one (in_j). Moreover, a special symbol δ causes the enclosing membrane to be dissolved. Membrane configurations are written like $\langle M_1 \mid a\,b\,c\ \langle M_2 \mid c\,d \rangle\rangle$ where the membrane M_1 contains the objects a, b, and c, and the membrane M_2, which in turn contains two objects, c and d. Formally, the usual definition of a P system with n membranes is a tuple

$$\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, i_o)$$

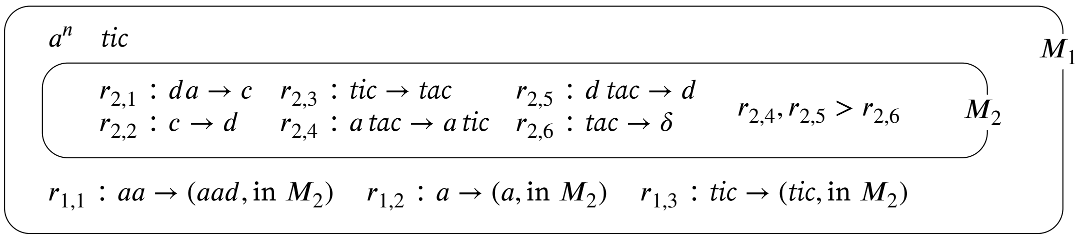

a^n tic
$r_{2,1} : da \to c$
$r_{2,2} : c \to d$
$r_{2,3} : tic \to tac$
$r_{2,4} : a\,tac \to a\,tic$
$r_{2,5} : d\,tac \to d$
$r_{2,6} : tac \to \delta$
$r_{2,4}, r_{2,5} > r_{2,6}$
M_2
$r_{1,1} : aa \to (aad, \text{in } M_2)$
$r_{1,2} : a \to (a, \text{in } M_2)$
$r_{1,3} : tic \to (tic, \text{in } M_2)$
M_1

Figure 8.1: Venn diagram of a divisor-calculator membrane system.

with the set O of objects, the initial contents w_i and the set of rules R_i of the n membranes, the index i_o of a membrane whose contents or cardinality should be considered as the result of computations, and the initial structure μ of the nested membranes, usually expressed as a tree or string of paired brackets. Membrane systems are usually represented graphically as Venn diagrams like Figure 8.1, which describes a system to compute divisors of a number, read as the number of d in the skin membrane. However, both the membrane contents and their structure will likely change during execution, so we will not usually explicit them aside from the configurations themselves. Note that this is a basic definition of P systems, and many variants have been proposed, either including special objects as promoters and inhibitors, allowing membranes to be created or duplicated, using more complex cell topologies like tissue-like and neural-like ones, etc. Some of these variants will be addressed in Section 8.5.

Membrane *computations* consist of successive applications of *evolution steps*. In turn, evolution steps are the parallel application of as many evolution rules as possible to the objects of each membrane, often regulated by priority relations. Irreducible configurations are those in which no evolution step is possible. More precisely, an evolution step consists of the following phases:

1. Applying the evolution rules to each membrane in a *maximal parallel* manner (see below).
2. Sending and receiving the objects contained in *out*, *in*, and *here* targets.
3. Dissolving membranes containing δ, thus dropping its objects and membranes to the enclosing membrane.

The *maximal parallel rewrite* step is described by a multiple choice of rules or multiset $A_i : R_i \to \mathbb{N}$ for each membrane M_i. A multiset of rules A *can be applied* to a multiset of objects W if the union of their lefthand sides with multiplicities is contained in W, and the result has that union replaced by the union of the righthand sides in A.

$$W \to_A W \setminus \bigcup_{(u\to v)\in A} u \ \cup \ \bigcup_{(u\to v)\in A} v$$

Such a choice A is *maximal* if $A + \{r\}$ cannot be applied to W for no matter which rule r.[1] In summary, a maximal parallel rewrite is the application of a maximal multiset of rules to each membrane $(A_1, \ldots, A_n)$ with at least one non-empty A_i. The choice of each A_i may not be unique, so this phase is nondeterministic.

When a priority relation ρ_i is imposed to a membrane, not all choices are *admissible*. Two ways of understanding rule priorities are considered:

- a *weak* sense, in which a choice A_i is admissible if for all $r \in R_i$ either $A_i(r') = 0$ for all $r >_{\rho_i} r'$ or the choice $A_i[r'/0]_{r >_{\rho_i} r'} + \{r\}$ cannot be applied;
- and a *strong* sense, in which a choice A_i is admissible if it is admissible in the weak sense and, in addition, $A_i(r') = 0$ for all $r >_{\rho_i} r'$ such that $A_i(r) > 0$.

[1]For multisets, we write $(A+B)(r) = A(r)+B(r)$, $(A-B)(r) = \max\{A(r)-B(r), 0\}$, $A[r/n](r) = n$ and $A[r/n](r') = A(r')$ if $r \neq r'$, and $\{r\}$ for the multiset with a single r.

Intuitively, in each step, the membrane objects in the configuration should be distributed among the membrane rules according to their priority relation. A rule should only be assigned objects if they cannot be used for higher priority rules. Using the objects left by them is possible in the weak sense, but disallowed in the strong sense. The parallel application of evolution rules is well described by a multiset of them, because the order in which they are applied does not matter, since they only remove chemicals from the membrane multiset and add others that cannot be used until the next evolution step. A relevant consequence is that the maximal parallel application of rule priorities in the weak sense can be calculated by the exhaustive sequential application of the rules where priorities are considered locally at each step.

For example, the divisor calculator of Figure 8.1 is intended to be executed from the initial configuration $\langle M_1 \mid a^n \, tic \, \langle M_2 \mid \, \rangle \rangle$ where a^n means n copies of a. The maximal parallel application of the rules in M_1 transfers in a single evolution step all the a and tic objects to M_2 along with a nondeterministic number of ds between 0 and $n/2$, which is the divisor candidate. These objects are actually communicated in the second phase of the evolution step according to the "in M_2" targets. The next steps take place in M_2, where the number of as is divided by the number of ds using successive subtractions that take two evolution steps each. When the tic object is present, $r_{2,1}$ removes an a for each d yielding a c, and $r_{2,3}$ turns the tic into a tac. In the next evolution step, the missing ds are recovered with $r_{2,2}$ and the rest of the subtraction is checked. If there are a objects left from the previous step, the division is not completed yet and $r_{2,4}$ transforms tac to tic for a new iteration. If there are d objects left, the number of as was not a divisor of the number of ds, so the execution is stopped by removing the tac object with $r_{2,5}$. Otherwise, there is neither as nor ds in the configuration, so the division is exact and a true divisor has been found. Then, the rule $r_{2,6}$ introduces the symbol δ to trigger the dissolution of M_2 in the third phase of the evolution step, whose objects are dropped to M_1. Notice that $r_{2,6}$ can only be applied according to the priorities if $r_{2,4}$ and $r_{2,5}$ cannot be applied, so that the membrane is not dissolved unless a divisor has been found. In this case, weak and strong priorities would produce the same result, since the application of $r_{2,4}$ or $r_{2,5}$ consumes the tac symbol, which impedes the execution of $r_{2,6}$.

8.2 Representing membrane systems in Maude

Membrane systems have already been specified in rewriting logic and in Maude [ACL07, AL09, ACL05, AC09]. Multisets of objects and the nested membrane structure are naturally represented by terms with commutative and associative operators, but the challenge is applying evolution rules locally and in a maximal parallel way. This has been solved in previous works by

- representing evolution rules as rewrite rules and controlling their application with the Maude reflective features [ACL05];
- by representing evolution rules as data and computing the steps at the object level [ACL07]
- or even using a particular combination of reflection and a primitive version of the Maude strategy language [AL09].

In either case, the specification of membrane configurations is very similar, and ours only slightly differs from those:

```
mod P-SYSTEM-CONFIGURATION is
  including QID-LIST .

  sorts Obj Membrane MembraneName Target TargetMsg .
  sorts EmptySoup MembraneSoup ObjSoup TargetSoup Soup .
```

```
   subsort  Obj < ObjSoup .
   subsort  Membrane < MembraneSoup .
   subsort  TargetMsg < TargetSoup .
   subsorts EmptySoup < MembraneSoup ObjSoup TargetSoup < Soup .

   op <_|_>   : MembraneName Soup → Membrane [ctor] .
   op delta   : → Obj [ctor] .

   op empty   : → EmptySoup [ctor] .
   op __      : Soup Soup → Soup [ctor assoc comm id: empty] .
```

A membrane is identified by a name and contains a multiset of juxtaposed objects, membranes, and targets of sort `Soup`. Each component defines a subsort to facilitate operating with them, `ObjSoup`, `MembraneSoup`, and `TargetSoup`. Several omitted operator declarations specify how these subsorts combine with the `__` operator. Target messages are expressed as pairs:

```
   ops here out : → Target [ctor] .
   op  in_ : MembraneName → Target [ctor] .
   op `(_,_`) : Soup Target → TargetMsg [ctor frozen (1)] .
```

Rewriting is proscribed in the first argument of the message with the `frozen (1)` annotation, since objects in messages have been generated by evolution rules and must not be used until the next evolution step takes place. Messages with the same target are combined into a common pair by an equation, and three rules are defined to resolve the communication between cells:

```
   vars MN MN'  : MembraneName .
   vars W W' CW : Soup .
   var  T       : Target .

   eq (W, T) (W', T) = (W W', T) .

   rl [here] : (W, here) ⇒ W .
   rl [in]   : (CW, in MN) < MN | W > ⇒ < MN | W CW > .
   rl [out]  : < MN | W (CW, out) > ⇒ < MN | W > CW .
```

Finally, another rule `dis` triggers the effect of the δ symbol by dissolving the non-skin membrane where it is contained. The skin or outermost membrane is never dissolved, as enforced by the nested pattern in the rule.

```
   rl [dis] : < MN | W < MN' | W' delta > > ⇒ < MN | W W' > .
 endm
```

This `P-SYSTEM-CONFIGURATION` module is the common and generic basis of the rewrite theories that will specify concrete membrane systems, where the `MembraneName` and `Obj` sorts are populated, and the evolution aspects are defined.

Evolution rules are represented as identical rewrite rules, delegating their controlled application to strategies. These strategies are partially generic and partially dependent on the rules and priorities of the membrane system, but never on any particular configuration. The membrane-specific strategy definitions are accessed through the strategy `handleMembrane` that takes the membrane name as argument. For a membrane M with rules $r_1, \ldots, r_n$ and without priorities, `handleMembrane` will be defined as the exhaustive application of these rules, gathered in an auxiliary strategy `membraneRules`[2]

sd membraneRules(M) := r_1 | ⋯ | r_n .

[2]Another possibility to limit the locality of evolution rules is adding a membrane context: the rule $v \to w$ for the membrane M could also be transformed into < M | v `S:Soup` > ⇒ < M | w `S:Soup` >.

The generic part is specified in the following module `P-SYSTEM-STRATEGY`, where the strategy `mpr` defines the maximal parallel step, in terms of the system-specific `handleMembrane`.

```
smod P-SYSTEM-STRATEGY is
  protecting P-SYSTEM-CONFIGURATION .

  strat  handleMembrane : MembraneName @ Soup .
  strats mpr visit-mpr communication @ Soup .
  strat  nested-mpr : MembraneSoup @ Soup .

  var MN : MembraneName .  var TM : TargetMsg .
  var S  : ObjSoup .       var TS : TargetSoup .
  var MS : MembraneSoup .  var K  : Nat .

  sd mpr := visit-mpr ; amatch TM ; communication ; (dis !) .

  sd communication := (in | out | here) ! .

  sd visit-mpr := matchrew < MN | S MS > by S  using handleMembrane(MN),
                                           MS using nested-mpr(MS) .

  sd nested-mpr(empty) := idle .
  sd nested-mpr(M MS) :=  (matchrew M MS by M using visit-mpr) ;
                          nested-mpr(MS) .
```

The three phases of an evolution step (see Section 8.1) are concatenated in the main strategy `mpr`: (1) `visit-mpr` applies the evolution rules to the membranes, (2) `communication` transmits all targeted objects through the membranes, and (3) `dis !` dissolves them exhaustively. The `visit-mpr` strategy executes the parameter `handleMembrane` on the objects of the topmost membrane, and then continues with the nested ones using `nested-mpr`. The requirement that at least an evolution rule must be applied in the whole system is enforced by the `amatch TM` test that discards an execution if no target has been generated in the whole configuration.[3] `nested-mpr` receives the full nested membrane soup and recursively processes the membranes one at a time. Note that the definition of `nested-mpr` is not efficient, since all possible matches of the set-like argument will be tried at each call, hence unnecessarily processing the membranes in all possible orderings. This will be prohibitive when dealing with exponential-size membranes in Section 8.5.2, where alternatives are discussed.

The `handleMembrane` strategy can be accommodated to different situations, depending on the rule priorities and their interpretation. The case without priorities or with priorities in their weak sense can be handled by the following strategy `inner-mpr`:

```
strat inner-mpr : MembraneName @ Soup .
sd inner-mpr(MN) := membraneRules(MN) ! .
```

For a membrane name `MN`, it calls `membraneRules` repeatedly until it cannot be applied again. Since the multiset argument of target messages is declared `frozen`, i.e. it is excluded from rewriting, products of the evolution rules are not used in the same step.[4] With `membraneRules` being a disjunction of rules, as above, the `mpr` strategy implements a maximal parallel step. The proofs of this statement and the other propositions in this chapter can be found in [RMP+22b].

Proposition 8.1. *The strategy* `mpr` *executes a maximal parallel evolution step without priorities when* `membraneRules(`M`)` *is defined as the disjuntion of all rules for* M*, i.e., under these conditions, a*

[3]In previous versions, `visit-mpr` and their auxiliary strategies take care themselves of whether at least a rule has been applied by using conditional operators and observing whether rules succeeded. However, looking at the presence of a target message is equivalent and simpler.

[4]An alternative to exclude rule products from being used again in the same step is separating loose objects from targets with a `matchrew` pattern `OS TS`. However, this is slower.

configuration C' is obtained by an evolution step from C iff C' is a result of the strategy `mpr` *applied on C.*

Priorities in the weak sense can also be handled by adding a lattice of `or-else` combinators to the `membraneRules` definition. Assuming that $P \subseteq R \times R$ is a generator set for this priority relation, the following recursive procedure is able to generate a strategy respecting the priorities at each application:

1. consider the disjunction $r_1 \mid \cdots \mid r_n$ of the minimal elements in P,
2. replace each such r by $(r_1 \mid \cdots \mid r_n)$ `or-else` r where $r_1, \ldots, r_n$ are all rules satisfying $(r_i, r) \in P$, and
3. iterate (2) on the newly introduced rules up to the maximal elements.

Expressions like α `or-else` $\beta \mid \alpha$ `or-else` γ can be simplified on the fly to α `or-else` $(\beta \mid \gamma)$ to optimize the algorithm. The correctness of this procedure follows from the fact that a rule will only be applied when its predecessors in the order have failed, due to the semantics of the `or-else` combinator, and that all rules appear in the expression because they must be reachable from the minimal elements of the order relation. The size of the strategy is bounded by the number of rules and the relation pairs in P. An example is shown in Figure 8.2.

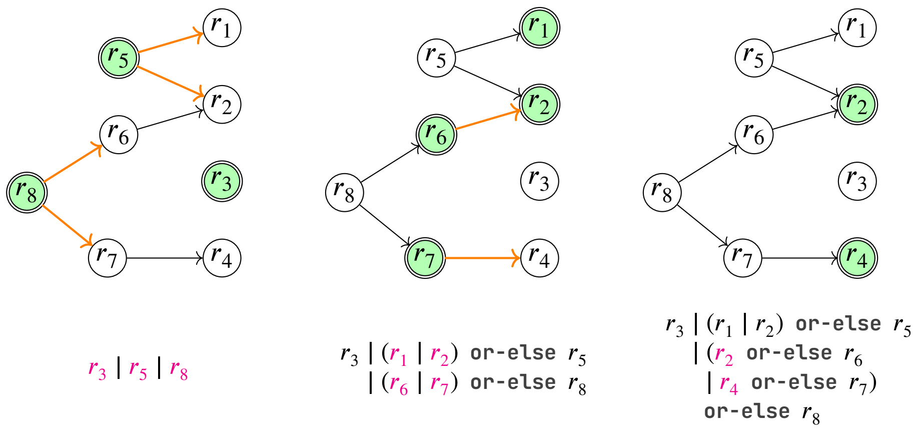

r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8
$r_3 \mid r_5 \mid r_8$
$r_3 \mid (r_1 \mid r_2)$ or-else r_5 $\mid (r_6 \mid r_7)$ or-else r_8
$r_3 \mid (r_1 \mid r_2)$ or-else r_5 $\mid (r_2$ or-else r_6 $\mid r_4$ or-else $r_7)$ or-else r_8

Figure 8.2: Strategy generation for an example weak priority relation.

Proposition 8.2. *The strategy* `mpr` *executes a maximal parallel evolution step with weak priorities when* `membraneRules(`M`)` *is defined as indicated above.*

When the priority of the rules is understood in the strong sense, the skeleton of `inner-mpr` cannot be exploited since it executes each rule independently in the sequence, and respecting strong priorities requires knowing which rules have already been applied. However, a variation of the previous procedure can be used. In this case, the parameter `handleMembrane` is defined using a new recursive strategy `strong-mpr` that receives the set of labels of the already applied rules:

```
strat strong-mpr : MembraneName QidSet @ Soup .
sd handleMembrane(MN) := strong-mpr(MN, empty) .
```

The definition of `strong-mpr(`M`, AR)` for a membrane M whose priority is generated by P is given recursively as:

1. Take the disjunction of α_r `:=` r `; strong-mpr(`M`, (`r`, AP))` for every minimal element of P. The recursive call adds r to the comma-separated set `AP` of applied rules.

2. Replace each element α_r in the disjunction by

   $(\alpha_{r_1}$ `|` $\cdots$ `|` $\alpha_{r_n})$ `or-else`
   `(match S s.t.` $\{r' \in R_i \mid r' >_{\rho_i} r\}$ `intersect AP = empty ;` $\alpha_r)$

   where `S` is a variable of sort `Soup`, and $r_1, \ldots, r_n$ are all the elements that satisfy $(r_i, r) \in P$. The test prevents the rule r from being applied if a rule with higher priority has already been used.

3. Iterate (2) on the newly introduced elements up to the maximal ones.

4. Take α `or-else idle` where α is the result of (3).

Because of the guards, the previous construction guarantees that rules are executed only if no rule with higher precedence has been applied.

Proposition 8.3. *The strategy `mpr` executes a maximal parallel evolution step with strong priorities when `handleMembrane` is instantiated to the `strong-mpr` strategy described above.*

Finally, executions up to irreducible configurations are described by the following strategy `mcomp`. Unlike in `mcomp2`, used to define it, trivial executions not taking any step are excluded.

```
strats mcomp mcomp2 @ Soup .
sd mcomp  := mpr ; mcomp2 .
sd mcomp2 := mpr ? mcomp2 : idle .
```

Computations up to a maximum number of steps or until a given number of objects is reached can also be specified with definitions like

```
sd mcomp(0) := idle .
sd mcomp(s(K)) := mpr ? mcomp(K) : idle .
sd mcomp-obj(K) := match S s.t. numObjsRec(S) >= K
                   or-else (mpr ? mcomp-obj(K) : idle) .
endsm
```

where `numObjsRec` recursively counts the number of objects in the given soup.

Fortunately, the interactive environment in Section 8.4 will make the manual instantiation of these strategies unnecessary: `membraneRules` and `strong-mpr` are constructed equationally at the metalevel following the above procedures from the membrane programs read from file.

8.3 Model checking temporal properties

Like any other strategy-controlled Maude module, membrane systems can be model checked using the tools presented in this thesis. To facilitate the specification of temporal properties, we provide a predefined set of atomic propositions that include among others:

```
mod P-SYSTEM-PREDS is
  protecting P-SYSTEM-CONFIGURATION .
  including SATISFACTION .
  protecting EXT-BOOL .

  subsort Soup < State .

  sort NatExpr BoolExpr .
  subsort Nat < NatExpr .
  subsort Bool < BoolExpr .
```

```
op isAlive  : MembraneName              → Prop [ctor] .
op contains : MembraneName Soup         → Prop [ctor] .
op contains : MembraneName MembraneName → Prop [ctor] .
op hasDelta :                           → Prop [ctor] .

op {_}      : BoolExpr          → Prop [ctor] .
op _=_      : NatExpr NatExpr   → BoolExpr [ctor] .
op _+_      : NatExpr NatExpr   → NatExpr [ctor] .
op count    : MembraneName Soup → NatExpr [ctor] .
```

The property `isAlive` checks whether a membrane is present in the configuration, and `contains` whether it contains some objects or membranes. More complex properties can be built with Boolean and integer expressions of sorts `BoolExpr` and `NatExpr` between curly brackets. The multiplicities of any multiset in any membrane, designated with the `count` operator, can be combined with the arithmetical operators and relations supported by Maude. For example, the property `{ count(M1, a) = 2 * count(M2, b) }` says that the number of `a`s in `M1` doubles the number of `b`s in `M2`. The evaluation of these expressions and the satisfaction of these propositions is defined equationally in the same module. For example, the command

```
red modelCheck(< M1 | a b < M2 | a > >, [] contains(M1, a), 'mcomp, 'mpr) .
```

checks whether `M1` always contains an `a` in all membrane executions from the given initial one. The system is controlled by the membrane computation strategy `mcomp` while considering the evolution step strategy `mpr` as an opaque strategy, so that the steps of the model are evolution steps. The interactive environment of the next section will do all this behind the scenes.

8.4 The membrane system environment

The executable rewriting logic framework proposed to represent membrane systems can be directly instantiated in Maude itself with a particular system: extending `P-SYSTEM-STRATEGY` with the declaration of its objects and its membranes, their evolution rules as rewrite rules, and their ascription to a membrane with strategies. However, dealing with priorities or other extensions is not so simple, but can be automatically done by convenient program transformations. The interactive environment described in this section implements these manipulations and allows simulating and verifying membrane systems easily. These should be specified in an extended version of the membrane description language of a previous prototype in [AL09], on which ours was initially based.

After loading the `memrun.maude` file of the interactive environment distribution into Maude, the following command will execute its interface:[5]

```
Maude> erewrite initREPL(repl) < repl : MemREPL | none > .

      ** Membrane system environment in Maude **

Membrane>
```

The environment offers different commands that are listed by typing `help`.

```
Membrane> help .
Available commands:
  load "filename"                           Loads a membrane system from a file
  [dfs] transition <membrane> .
  [dfs] trans <membrane> .                  Simulates a single evolution step
```

[5]In Maude 3.1, file operations are disabled by default for security reasons, so `-allow-files` must be given as a command line argument to Maude in order to use the environment.

```
  [dfs] compute <membrane> .                Simulates a membrane computation
  check <membrane> satisfies <formula> .    Model check the formula
  show membranes .                          Show the current membranes names
  show <membrane name> .                    Show the membrane definition
  set priority [weak | strong] .            Set how priority is understood
  quit .                                    Quits the interpreter
```

The `load` command reads a membrane specification from a file and runs the commands in it. For example, `load divisor.memb` loads the membrane system of Figure 8.1.

```
Membrane> load divisors.memb
File divisors.memb has been loaded.
```

In that file, evolution rules and priorities are specified as shown below for the membrane `M2`:

```
membrane M2 is
  ev r21 : d a   → c .         ev r22 : c     → d .
  ev r23 : tic   → tac .       ev r24 : a tac → a tic .
  ev r25 : d tac → d .         ev r26 : tac   → delta .
  pr r24 > r26 .
  pr r25 > r26 .
end
```

Loading the file implies generating the strategies described in Section 8.2 for later use by the various supported commands. They can be shown with the `show strats` command followed by the membrane name.

```
Membrane> show strats M2 .
Weak priority: (r24 | r25 or-else r26) | r21 | r22 | r23
Strong priority: r21 ; mpr-strong(M2, ('r21, AR))
  | r22 ; mpr-strong(M2, ('r22, AR))
  | r23 ; mpr-strong(M2, ('r23, AR))
  | (r24 ; mpr-strong(M2, ('r24, AR))
    | r25 ; mpr-strong(M2, ('r25, AR))
  or-else match H s.t. intersection(('r24, 'r25), AR) =
          empty ; r26 ; mpr-strong(M2, ('r26, AR)))
```

If the command omits the `strats` word, the membrane definition is shown instead, and `show membranes` displays the names of all loaded membranes.

The `trans` and `compute` commands allow simulating evolution steps and computations. The first one executes a single step, indicating the multiset of rules applied for each membrane.

```
Membrane> trans < M1 | a a a tic < M2 | d tac > > .
Solution 1 with r11 r12 r13 in M1, r25 in M2 :
  < M1 | c c c < M2 | a a a d d tic > >
Solution 2 with r12 r12 r12 r13 in M1, r25 in M2 :
  < M1 | c c c < M2 | a a a d tic > >
No more solutions.
```

The `compute` command shows all irreducible states that can be found by successive transitions.

```
Membrane> compute < M1 | a a a a a a a a tic < M2 | empty > > .
Solution 1: < M1 | d d d d >
Solution 2: < M1 | < M2 | d d d > >
Solution 3: < M1 | d d >
Solution 4: < M1 | d >
No more solutions.
```

For this divisor calculator, the solutions tell us that 2 and 4 are the non-trivial divisors of 8, after reading their number of ds in M1. The interpretation of rule priorities, either weak or strong, can be set globally with the set priority command that changes the definition of the handleMembrane strategy mentioned in Section 8.2. By default, strong priorities are used.

Temporal properties of the membrane executions can be checked with the check command. These properties are expressed in LTL for the predefined language of atomic propositions described in Section 8.3, which can anyhow be extended by modifying the environment source code. For example, the following command checks that the number of ds in the membrane M1 is either 0 or a divisor of 12 in all reachable configurations from an initial membrane with 12 as. Thus, a false divisor is never generated.

```
Membrane> check < M1 | a a a a a a a a a a a a tic < M2 | empty > >
          satisfies [] ({ count(M1, d) = 0 } \/ { count(M1, d) divides 12 }) .
The property is satisfied.
```

When the property is not satisfied, the output shows a counterexample describing the intermediate steps and the rules that have been applied. This is the case of the following property claiming that every state containing tac in M2 is followed by a state containing tic.

```
Membrane> check < M2 | a a d d tic >
          satisfies [] (contains(M2, tac) -> O contains(M2, tic)) .
| < M2 | a a d d tic >
v with r21 r21 r23 in M2
| < M2 | c c tac >
v with r22 r22 r26 in M2
X < M2 | delta d d >
```

The check command admits bounded model checking on the number of objects in the configuration, where the bound may be indicated between brackets after the check keyword. This is useful for membrane systems that, unlike this example, have an unbounded configuration space. For instance, the following membrane system calculates the squares $\langle M_1 \mid d^n\, e^{n^2} \rangle$ of all natural numbers $n \geq 1$ starting from $\langle M_1 \mid \langle M_2 \mid \langle M_3 \mid a\, f \rangle\rangle\rangle$.

```
membrane M1 is end

membrane M2 is
  ev r21 : b   -> d .        ev r22 : d -> d e .
  ev r23 : f f -> f .        ev r24 : f -> delta .        pr r23 > r24 .
end

membrane M3 is
  ev r31 : a -> a b .        ev r32 : a -> b delta .      ev r33 : f -> f f .
end
```

The membrane M3 nondeterministically produces a number $n \geq 1$ of bs along with 2^n fs in n steps, and then spills its contents into M2. Then M2 generates one e for each d in every one of the n steps required to reduce the exponential number of fs with r23, hence calculating n^2.

```
Membrane> load nsquare.memb
File nsquare.memb has been loaded.
Membrane> compute [3] < M1 | < M2 | < M3 | a f > > > .
Solution 1: < M1 | d e >
Solution 2: < M1 | d d e e e e >
Solution 3: < M1 | d d d e e e e e e e e e >
No more solutions requested.
```

The (not so) atomic proposition { count(M1, d) ^ 2 = count(M1, e) } claims that the number of es is the square of the number of ds in M1. This property cannot be checked as an invariant on the whole infinite state space, but it can be, for instance, in the reachable portion of the model where the number of objects is always below 70.

```
Membranes> check [70] < M1 | < M2 | < M3 | a f > > >
           satisfies [] { count(M1, d) ^ 2 = count(M1, e) } .
The property is satisfied.
```

Moreover, membrane systems can also be checked against CTL* and μ-calculus properties, using the external model checkers for strategy-controlled systems [RMP+20c]. Since these are not integrated in the Maude interpreter, they should be used through an external tool instead of the membrane environment. For instance, resuming the divisor calculator example, we can check the μ-calculus property $\nu Z.\, \mathit{isAlive}(M_2) \wedge [\cdot](\neg\, \mathit{isAlive}(M_2) \vee [\cdot]\, Z)$ asserting that the membrane M_2 is only dissolved in odd steps.

```
$ ./membranes.py -v divisors.memb \
   '< M1 | a a a a a a a a a a a a tic < M2 | empty > >' \
   'nu Z . (isAlive(M2) /\ [.] (~ isAlive(M2) \/ [.] Z))'
Rewriting model generated in 4101 rewrites.
The property is satisfied (70 states, 76884 rewrites).
```

In fact, the first clause $\mathit{isAlive}(M_2)$ claims that M_2 is present in the configuration, and the second one requires that after any possible step $\neg\, \mathit{isAlive}(M_2) \vee [\cdot]\, Z$ is satisfied, meaning that either M_2 is no longer present or in the next step the property holds again, as mandated by the fixed point ν. This property cannot be expressed in CTL*. Checking whether at least a divisor of 12 is found can be done with the following CTL property, although this is also possible by simulating with the `compute` command.

```
$ ./membranes.py divisors.memb \
   '< M1 | a a a a a a a a a a a a tic < M2 | empty > >' \
   'E <> { count(M1, d) divides 12 }'
The property is satisfied (70 system states, 91136 rewrites).
```

Model checking in a bounded subset of the state space is also possible with this interface, both on the number of objects and on the number of evolution steps.

8.4.1 Implementation notes

This prototype is based on the Maude specification of membrane systems in Section 8.2, and uses Maude reflection to construct the rules and strategies of the specific membrane systems parsed from the input. The external objects introduced in Maude 3 are used to read commands and write their results to the terminal, and also to read membrane specifications from external files. In either case, text is read as a string and parsed using the `metaParse` descent function in specific modules describing the grammar of the different elements, as usual in Maude metalanguage interfaces.

Evolution rules are directly translated into rewrite rules, but loose objects in the righthand side are enclosed into a `here` target, because targets are used to exclude consumed terms from being rewritten twice in the same evolution step. For some commands, the righthand side of rules is appended a `log` object with its label to get track of the rules that have been applied, without interfering with the execution of the system. This allows showing them in the `trans` command and the model-checking counterexamples.

For example, the following equation is the actual translation from parsed evolution rules (containing unparsed fragments, known as bubbles) in the membrane specification to Maude rewrite rules at the metalevel.

```
ceq makeRules(M, 'ev_:_-->_.['token[Q], 'bubble[LHS], 'bubble[RHS]]) =
      (rl PLHS => PRHS [label (downTerm(Q, 'UNNAMED))] .)
  if PLHS := getTerm(metaParse(M, none, downTerm(LHS, (nil).QidList), 'ObjSoup))
```

```
   /\ T    := getTerm(metaParse(M, none, downTerm(RHS, (nil).QidList), 'Soup))
   /\ PRHS := getTerm(metaReduce(M, 'wrapHere[T])) .
```

The module `M` would be an extension of `P-SYSTEM-CONFIGURATION` (see Section 8.2) populated with the inferred signature of objects for the particular membrane system. The lefthand side is parsed in the `ObjSoup` sort of this module, while the righthand side is a `Soup` allowed to contain targets. The function `wrapHere` wraps the free objects of the righthand side into a `here` target, as explained before.

After the membrane specification is parsed, strategies are generated for the prioritized versions of the maximal parallel step. For instance, the procedure to generate the weak priority strategy explained in Section 8.2 is executed by a fixed-point equational computation that starts with the minimal elements of the relation

```
op genPriorityStrat : QidSet PriorityRelation → Strategy .
op genPriorityStrat : Strategy PriorityRelation → Strategy .

eq genPriorityStrat(Rs, PR) =
     genPriorityStrat(genRuleApps(minimal(Rs, PR)), PR) .

op genRuleApps : QidSet → Strategy .
eq genRuleApps(none) = fail .
eq genRuleApps(R ; Rs) = (R[none]{empty}) | genRuleApps(Rs) .
```

and iterates extending the newly introduced rules until the strategy is stabilized.

```
eq genPriorityStrat(S, PR) =
   if orElseSimplify(extendPrec(S, PR)) == S then S
   else genPriorityStrat(orElseSimplify(extendPrec(S, PR)), PR) fi .

op extendPrec : Strategy PriorityRelation → Strategy .

eq extendPrec(S1 or-else S2, PR) =
     extendPrec(S1, PR) or-else S2 .
ceq extendPrec(S1 | S2, PR) = extendPrec(S1, PR) | extendPrec(S2, PR)
 if S1 =/= fail /\ S2 =/= fail .
eq extendPrec(R[none]{empty}, PR) = if pred(R, PR) == none
   then R[none]{empty}
   else genRuleApps(pred(R, PR)) or-else R[none]{empty} fi .
```

where `pred` returns the predecessors of a rule in the priority relation `PR`. The strategy for strong priorities is generated similarly. Reflective module transformations assign the adequate `handleMembrane` definition depending on how rule priorities are understood.

The `trans` and `compute` commands rewrite the membrane term with `metaSrewrite` using the strategies `mcomp` and `mpr`, respectively, while the `check` command essentially reduces a `modelCheck` term like the one shown in Section 8.3 using `metaReduce`. The command line utility to model check against branching-time properties generates the membrane system theory using the same equational infrastructure of the interactive environment. However, instead of using Maude external objects for reading files and printing messages, the standard Python library is used, in which it is programmed. Once the model is set up, it is a strategy-controlled Maude specification that is directly passed to the unified model-checking tool `umaudemc` in Chapter 4 via its Python API.

8.5 Variations of membrane systems

Many variants of membrane systems have been proposed to better address different applications and problems [Pău02, CPP06]. The flexibility of the Maude language and its strategies makes modifying the prototype to support and experiment with these variations a relatively easy task. In this section, we illustrate this extensibility with three widespread features. Each subsection starts by describing and motivating one of the variants, then explains how it is implemented in the prototype, and finally shows the feature in action with an example. In Section 8.5.2, we also discuss how to fix the order in which membranes are processed to avoid redundant calculations, as anticipated in previous sections.

8.5.1 Structured objects

Standard membrane systems operate on multisets of unstructured opaque objects, but chemicals in a cell are usually complex molecules (DNA, proteins, ...) which are better described by structured data like strings or trees. *String rewriting P systems* [CV07, CPP06] are membrane systems whose objects are strings made out of terminal and nonterminal symbols, and whose evolution rules have the form $A \to w$ where A is a nonterminal symbol and w is a word. Targets are expressed similarly. Evolution rules are applied like in the standard P systems, but only one rule can be applied to each string in the membrane soup at each evolution step.

A generalization of this possibility has been implemented in the rewriting logic framework presented here, where the language of objects of the membrane system consisted of some plain identifiers, implicitly declared as they are used in the membrane specification. In this section, the user will be allowed to explicitly declare the signature of objects, which may include arbitrary Maude terms in their arguments to be considered modulo equations and axioms. Specific syntax in the membrane specification language is devoted to the definition of objects. For instance, the following lines declare string objects on some constants `a`, `b`, and `c`. The associative binary operator `_·_` stands for string concatenation, which is associative, and `eps` is the empty word.

```
signature is
  ob _·_ : Obj Obj [assoc id: eps prec 30] .
  obs a b c eps .
end
```

Object-declaration statements are similar to regular Maude operator declarations except that the range sort is omitted and that they are introduced by the **`ob`** or **`obs`** keywords. The sort `Obj` of objects is the implicit range of all object declarations. Maude functional modules may be imported with the `import` statement, as shown in the example of the following section. Orthodox string rewriting P systems can be defined on top of this signature by using only evolution rules of the form $A \to u$, but different signatures and other types of rules can be tried instead.

The technical difficulty of applying rules in structured objects is ensuring that each object is only rewritten once in each step, as required for strings. Moreover, if evolution rules were translated as in the basic model, they could get applied on the arguments of structured objects with undesired results, like targets appearing inside objects. Hence, the different types of rules have to be considered carefully. Rules $r : u \to (v_1, m_1) \cdots (v_n, m_n)$ with multiple targets m_k can only be applied at the top multiset, so they are executed with the strategy

```
matchrew O R by O using top(r)
```

where `O` and `R` are variables of sorts `Obj` and `Soup`. Any rule with a single target $r : t \to (t', m)$ can in principle be applied anywhere, either at the top multiset or at any position p of an object o. If t matches o in p with substitution σ, then the object o must become the target $(o[p/\sigma(t')], m)$. This is achieved by transforming the evolution rule r to the rewrite rule $r' : t \to t'$, and wrapping the object with the appropriate target once rewritten using the auxiliary rule `wrapMsg`:

```
matchrew O R by O using r' ; wrapMsg[M <- m]
rl [wrapMsg] : R => (R, M) [nonexec] .
```

Extending the membrane specification language, we add the declaration statement **`xev`** for those rules that are meant to be applied inside objects, which can have at most one target.

```
xev lbl : t → (t', m) .
```

Rules declared with **ev** are applied only at the top even if they can match inside an object. Rules whose lefthand side is a multiset are not wrapped for efficiency, because object multisets are assumed to only appear at the top level.

The following simple membrane system defines three evolution rules on the object string signature presented before.

```
membrane M1 is
  xev s1 : a · a → a .
  xev s2 : b → c · c .
  ev  s3 : a → c .
end
```

The rule `s1` removes duplicated `a`s in strings, while `s2` transforms each `b` in a pair of `c`s. The last rule `s3` transforms loose `a` objects into `c`, but it is not applied on the characters of a string because it uses the **ev** instead of the **xev** keyword.

```
Membrane> load strings.memb
File strings.memb has been loaded.
Membrane> trans < M1 | (a · a · b · a) b (a · a) > .
Solution 1 with s1 s1 s2 in M1 :
  < M1 | (a · b · a) (c · c) a >
Solution 2 with s1 s2 s2 in M1 :
  < M1 | (a · a · c · c · a) (c · c) a >
No more solutions.
Membrane> compute < M1 | (a · a · b · a) b (a · a) > .
Solution 1:   < M1 | (a · c · c · a) (c · c) c >
No more solutions.
```

The `show strats` command lets us observe that the `membraneRules` strategy is generated as explained above.

```
Membrane> show strats M1 .
Weak priority:  matchrew O R by O using top(s3)
   | matchrew O R by O using(s1 ; top(wrapMsg[T ← here]))
   | matchrew O R by O using(s2 ; top(wrapMsg[T ← here]))
```

Another example with structured objects, but without subterm rewriting, is shown in the following section.

8.5.2 Membrane division

Another important and popular extension of membrane systems is membrane division, inspired on the cellular *mitosis*, that allows membrane systems to grow and take advantage of the parallelism of the computation model. In our realization, membrane division is triggered by a new target that replaces the affected cell by two copies of itself with all of its contents, plus a different set of objects for each copy included in the target triple. The following declaration and rule should be added to `P-SYSTEM-CONFIGURATION`:

```
op `(_,_,div`) : ObjSoup ObjSoup → TargetMsg [ctor frozen (1 2)] .

rl [div] : < MN' | EW < MN | CW (W, W', div) > >
        ⇒ < MN' | EW < MN | CW W > < MN | CW W' > > .
```

Division is done by the new rule `div` above, which prevents duplicating the outermost membrane. This rule should be applied exhaustively in the third phase of the evolution steps, just after object communication has been completed, but before membranes are dissolved. This involves changing the `mpr` strategy definition:

```
sd mpr := visit-mpr ; amatch TM ; communication ; (div !) ; (dis !) .
```

Other cell operations like creation, merge, endocytosis (introducing a membrane into another one), exocytosis (expelling a membrane), and gemmation could be implemented similarly.

Combining both membrane division and structured objects, the following membrane system implements a Boolean satisfiability (SAT) solver that runs in a polynomial number of evolution steps on the size of the formula. Each logical variable triggers a membrane duplication where each copy evaluates a possible value of the variable, making the membrane grow exponentially to evaluate in parallel all possible valuations. Formulae are specified using acyclic graphs indexed by natural numbers from the Maude's `NAT` module, with 0 being the root of the formula. Each node is given by a symbol whose first argument is its own identifier, followed by the values or identifiers of the node arguments. The role of the `splitoken` object will be explained later.

```
signature is
  import NAT .

  ob  const  : Nat Bool .                        *** Logical constant
  ob  var    : Nat .                                    *** Variable
  ob  not    : Nat Nat .                                *** Negation
  obs and or : Nat Nat Nat .                    *** Binary operators

  ob splitoken .                          *** Token to limit splitting
end
```

For example, $x \wedge \neg x$ can be written `and(0, 1, 2) var(1) not(2, 1)`.

The SAT solver consists of two nested membranes `M1` and `M2`. The skin membrane `M1` is the output membrane of the SAT solver, where the presence of an object `const(0, true)` indicates the satisfaction of the formula. However, we define it empty **membrane** `M1` **is end** as a mere receptor of the objects from `M2`. The `M2` membrane includes several rules to simplify the expressions, a rule `split` to fork the membrane with two copies where a variable takes alternatively the `true` and `false` values, and a rule `end` that dissolves the membrane when the evaluation has finished. Variables can be declared with the **var** statement as in Maude and used in the evolution rules. In this case, they match the integer indices and Boolean constants in the nodes.

```
membrane M2 is
  var H M N : Nat .
  var B     : Bool .

  ev split : var(H) splitoken
         → (const(H, true), const(H, false), div) splitoken .

  ev not   : not(H, N)        const(N, B)
         → const(H, not B)  const(N, B)                    .
  ev and1  : and(H, M, N)     const(M, false)
         → const(H, false)  const(M, false)                .
  ev and2  : and(H, M, N)     const(N, false)
         → const(H, false)  const(N, false)                .
  ev and3  : and(H, M, N)     const(M, true)   const(N, true)
         → const(H, true)   const(M, true)   const(N, true)  .
  ev or1   : or(H, M, N)      const(M, true)
         → const(H, true)   const(M, true)                 .
  ev or2   : or(H, M, N)      const(N, true)
         → const(H, true)   const(N, true)                 .
  ev or3   : or(H, M, N)      const(M, false)  const(N, false)
          → const(H, false)  const(M, false)  const(N, false) .

  ev end   : const(0, B) → const(0, B) delta .

  pr not and1 and2 and3 or1 or2 or3 end > split .
 end
```

Note that the split rule requires the object splitoken to be applied. This way, the number of cell divisions in each evolution step is limited to the number of such tokens, and some unnecessary cell divisions can potentially be avoided thanks to the simplification rules applied in the meanwhile.

The specification of this membrane system can be improved in many ways. For example, the and and or connectives are commutative, and so they can be described using a Maude-defined commutative pair that would reduce the number of rules. Moreover, each constant can only be used once in each evolution step as they are consumed by the evolution rule. This can be improved using promoters, which are introduced in Section 8.5.3. Finally, and more importantly, the choice of the next variable to be assigned is nondeterministic. Since the order in which variables are assigned does not affect the satisfaction of the formula, this unnecessarily and exponentially increases the size of the state space of the strategic execution. The depth-first variant of the compute command dfs compute is more convenient for evaluating this system than the default breadth-first search variant.

As advanced in Section 8.2, the strategy in charge of visiting the nested membranes of the configuration structure, nested-mpr, evaluates them in all possible orders, since all possible matches of the set-like argument will be tried at each call. These orders are factorially-exponentially many in the last example while the order in which membranes are processed is irrelevant (at least for the classes of membrane systems here considered). Hence, we must avoid it by fixing an order on the membranes, even at the expense of clarity.

```
op orderMembranes : MembraneSoup → MembraneList .
eq orderMembrane(empty) = nil .
eq orderMembrane(M MS) = insert(orderMembrane(MS), M) .

op insert : MembraneList Membrane → MembraneList .
eq insert(nil, M) = M .
eq insert(M1 ++ ML, M) = if lt(M, M1) then M ++ M1 ++ ML
                         else M1 ++ insert(ML, M) fi .
```

Using the orderMembranes function, nested-mpr can take a list of membranes (here assembled with the ++ symbol) instead of a set, and so process the membranes in a single fixed order. Membranes are ordered here using the lt operator of the TERM-ORDER module included in the Maude distribution that allows comparing arbitrary terms. However, this sorting algorithm is not really efficient and it is executed each time a membrane is processed. Sacrificing the due confluence modulo axioms of equational specifications, orderMembrane can be defined by the sole equation below, which will operationally fix the order in which the implementation visits the set.

```
eq orderMembrane(M MS) = M ++ orderMembranes(MS) .
```

Unfortunately, while this situation is frequent and the Maude strategy language includes an operator **one** that stops when the first solution is found, there is no builtin support for stopping at the first match in strategy calls or **matchrew**s.

For example, the SAT membrane system can be used to check the formulae $x \wedge \neg x$ and $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$. The const(0, true) will not be found in the solution of the first one because it is unsatisfiable, but it will be for the second formula.

```
Membrane> load sat.memb
File sat.memb has been loaded.
Membrane> compute < M1 | < M2 | splitoken and(0, 1, 2) var(1) not(2, 1) > > .
Solution 1: < M1 | const(0, false) … >
No more solutions.
Membrane> dfs compute [1] < M1 | < M2 | splitoken and(0, 5, 6) var(1) var(2)
                                        not(3, 1) not(4, 2)
                                        or(5, 1, 2) or(6, 3, 4) > > .
Solution 1:    < M1 | const(0, true) … >
No more solutions.
```

The second formula is evaluated quickly but not immediately with `compute`, so `dfs compute [1]` is used to find the first solution by depth. Additional evolution rules could be defined in `M1` and `M2` to clean the irrelevant objects and to allow recovering the valuation.

8.5.3 Promoters and inhibitors

Standard evolution rules only depend on the objects appearing in their lefthand side, which are consumed in its application. However, inspired in biochemical reactions, some processes may only take place in the presence of certain objects that are not part of the reaction. Conversely, some objects may act as inhibitors that impede a reaction to take place. This leads to rules with *promoters* and *inhibitors* [IS04, CPP06]. The general form of these rules is $u \to v \mid_z$ and $u \to v \mid_{\neg z}$, meaning that the usual evolution rule $u \to v$ can only be applied if the objects z distinct from u are present (respectively not present) in the membrane. The objects z are not consumed and can be used in the same evolution step.

Promoters and inhibitors have already been considered in rewriting logic and Maude [AC09]. The authors distinguish two semantics depending on whether promoters and inhibitors are allocated statically or dynamically. Static allocation corresponds to the description of this feature in the previous paragraph, and to the theoretical simultaneity of rule application. Dynamic allocation is more easily expressed in rewriting logic, where evolution rules are applied sequentially. It checks the presence of promoters and inhibitors on the intermediate state of the rewrite sequence, when some rules have already been applied and consumed part of the membrane contents. Their executable implementation in Maude only supports the latter, while we will implement the more widespread static allocation.

Given a rule $u \to v$ with promoters p and inhibitors h, we generate a conditional rewrite rule of the form

```
crl u ⇒ v if u p SRest := S0 /\ not contains(u h, S0) [nonexec] .
```

where the free variable `S0` occurs. When the rule is applied using its initial substitution, `S0` is instantiated with the contents of the membrane at the beginning of the evolution step. These contents are matched against a pattern that includes the promoters, the rule lefthand side and possibly something else by the first matching condition `:=`. In case the promoter objects contain free variables, these are instantiated and the rule is executed for all their possible matches. Inhibitors cannot bind new variables, being objects that are not present in the configuration, so they are handled by the equationally-defined `contains` predicate. This function decides whether its first argument is contained in the second argument, i.e. whether both the rule lefthand side and the inhibitors are included in the initial contents. A more strategic solution would have defined $u \to v$ directly and preceded its application by the tests `match W s.t. contains(u p, W0) /\ not contains(u h, W0)`, but u may contain variables that must take the same values in the test and the rule application, thus complicating the correct definition of the strategy. Changes need also be made to the main strategies, which should pass the initial membrane contents to the evolution rule applications.

```
sd visit-mpr := matchrew < MN | S MS > by S  using handleMembrane(MN, S),
                                     MS using nested-mpr(MS) .
sd handleMembrane(MN, S0) := inner-mpr(MN, S0) .
sd inner-mpr(MN, S0) := (matchrew S TS by S using membraneRules(MN, S0))
                        ? inner-mpr(MN, S0) : idle .
sd membraneRules(M_i, S0) := r_1 | ... | r_n | r'_1[S0 <- S0] | r'_m[S0 <- S0] .
```

The initial multiset is matched by the `S` variable of the `matchrew` operator in the `visit-mpr` strategy, and then, it is passed through strategy arguments to the definitions where rules with promoters and inhibitors are applied using the `[S0 <- S0]` initial substitution. Similar modifications are suffered by the strategies implementing the prioritized application of rules.

In order to allow expressing rules with promoters and inhibitors, the membrane specification language has been extended with a new statement that starts with the **cev** keyword and finishes with the specification of promoters and inhibitors after the **with** or **without** keyword (both or only one can be used). The multiset h may contain any variable that occurs in the lefthand side of the rule or in p.

```
cev lbl : u -> v with p without h .
```

Moreover, this syntax and the transformation described before can be easily extended to support more complex conditions or guards for evolution rules, like those used in *kernel P systems* [GI14] that include integer expressions on the multiplicity of arbitrary subsets in the initial multiset, in which promoters and inhibitors can be expressed.

Using this new feature, the SAT solver system of the previous subsection can be simplified and made more efficient. While the signature and the overall structure of the rules is kept unchanged, the common terms in both sides of evolution rules are placed as promoters, so that they can be used more than once in each step. In addition, the rule `split` on the variable `H` is inhibited by the object `var(s(H))`, hence forcing the variables to be assigned in decreasing index order (assuming they are numbered consecutively), and so limiting nondeterminism.

```
membrane M1 is end

membrane M2 is
  var H M N : Nat .
  var B     : Bool .

  cev split : var(H) splitoken -> splitoken
              (const(H, true), const(H, false), div)
              without var(s(H)) .

  cev not  : not(H, N)    -> const(H, not B)  with const(N, B) .
  cev and1 : and(H, M, N) -> const(H, false)  with const(M, false) .
  cev and2 : and(H, M, N) -> const(H, false)  with const(N, false) .
  cev and3 : and(H, M, N) -> const(H, true)   with const(M, true)
                                                   const(N, true) .
  cev or1  : or(H, M, N)  -> const(H, true)   with const(M, true) .
  cev or2  : or(H, M, N)  -> const(H, true)   with const(N, true) .
  cev or3  : or(H, M, N)  -> const(H, false)  with const(M, false)
                                                   const(N, false) .

  ev end   : const(0, B) -> const(0, B) delta .

  pr not and1 and2 and3 or1 or2 or3 end > split .
end
```

This membrane specification can be executed to check that the propositional formula $(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$ is satisfiable:

```
Membrane> load sat_promoters.memb
File sat_promoters.memb has been loaded.
Membrane> dfs compute [1] < M1 | < M2 | splitoken var(1) var(2) var(3) var(4)
                                        not(5, 3) not(6, 2) not(7, 4)
                                        or(8, 1, 5) or(9, 6, 3) or(10, 9, 7)
                                        and(0, 8, 10) > > .

Solution 1: < M1 | const(0, true) … >
```

It can be observed using the trans command that the evolution of the membrane system is deterministic, finishes in 8 evolution steps, and uses up to 16 simultaneous M2 membranes. The whole execution can be seen with the check command and the [] ~ contains(M1, const(0, true)) property.

8.6 Performance comparison

In this strategy-based rewriting framework for membrane systems, evolution steps are executed by exhaustively applying rules on each membrane, freely or according to some priorities. This involves visiting potentially many intermediate states for every admissible multiset of the membrane rules. Computing maximal parallel steps cannot completely avoid this, but more efficient algorithms are feasible by better planning the use of objects and compatible rules. For example, if the lefthand sides of two rules are disjoint multisets, all their interleaved applications will produce the same result, and so one could be executed exhaustively before the other.

We have compared other Maude-based simulators and model checkers for membrane systems with ours. First, the prototype of the work *Strategy-based proof calculus for membrane systems* by O. Andrei, and D. Lucanu [AL09] has been adapted to work with the current version of Full Maude, and the examples included in its distribution have been executed several times and measured with both prototypes. The only available command at their prototype is trans, and there are some bugs in its implementation that do not affect our comparison but prevent us from testing it with the other examples written for our prototype. On average, the new prototype is 9.47 times faster than the older, or 8.8 times if the initialization time of Maude and the prototypes is subtracted. Table 8.1 shows the execution times and their quotients for each example.

	Time (ms)					
Prototype	ex1	ex2	ex3	ex4	divisors	nsquare
Strategy-based... [AL09]	897	824	1014	851		
Executable... [ACL05]					1786	2945
This one	119	45	235	72	166	124
Speed-up	9.45	9.07	10.45	8.9	10.76	26.47
Without init	9.12	5.14	15.65	5.29	20.76	77.03

Table 8.1: Comparison with previous Maude-based prototypes.

Moreover, this prototype has also been compared with that of the work *Executable Specifications of P Systems* by O. Andrei, G. Ciobanu, and D. Lucanu [ACL05] with support for model checking. In this case, the divisors and nsquare examples have been checked against the properties discussed in Section 8.4.[6] The divisor calculator was translated to the language of their prototype, and the square number generator is the example included in their distribution.

[6]The performance comparison has been done with the property saying that the number of ds is a divisor of 12 in the divisors case, and the property saying that the number of ds is the square of the number of es in nsquare.

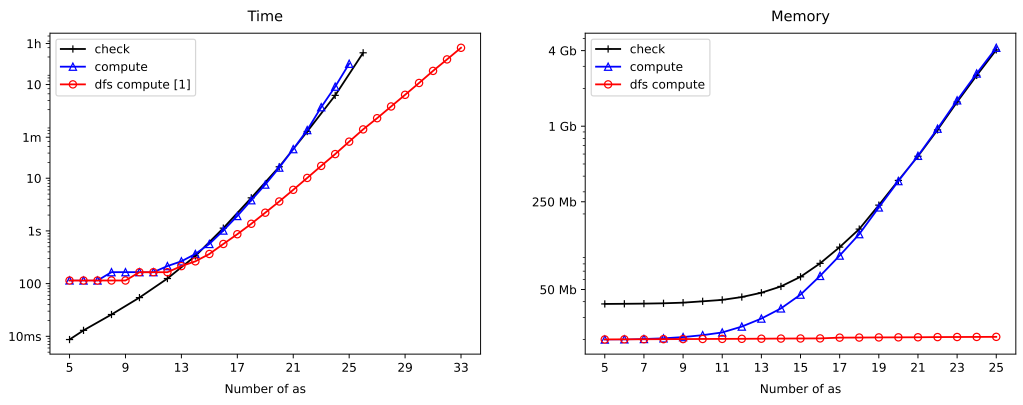


Figure 8.3: Execution time and memory usage for different commands on the divisor calculator.

This latter example is model checked in a bounded state space, which was fixed to 15 objects in their example and to 70 in Section 8.4. Since a limit of 70 takes much time in their prototype (it has been canceled after 5 minutes, while ours finishes in 2.5 seconds), a bound of 35 objects was fixed for both.

Nevertheless, if only the limit of 70 objects is increased to 71 in the previous property, the `check` command in our prototype does not finish within an hour. Similarly, we have pushed the capabilities of our prototype to the limit with the other examples in this paper. For the `divisors` calculator, we have checked the μ-calculus property in Section 8.4, computed all irreducible configuration terms with `compute`, and a single solution by depth with `dfs compute` from the initial term $\langle M_1 \mid a^n\ tic\ \langle M_2 \mid \rangle\rangle$ on increasing n. As shown in Figure 8.3, the execution time and the memory usage grow exponentially.[7] The only exception is the constant memory usage of the depth-first search `compute` command. Within an hour, results are obtained by `compute` and `check` for $n \leq 25$, and by the depth-first search of a single solution for $n \leq 33$. However, the results for $n \leq 23$ and $n \leq 28$ respectively have already been obtained in five minutes. For the SAT solver with promoters and inhibitors in Section 8.5.3, the depth-first search can stand up to 15 distinct variables in 5 minutes and to 17 in an hour.

Regarding the different extensions in Section 8.5, the performance penalty caused by them with respect to the functionality of the basic prototype is relatively small, since these features only produce an additional cost when they are effectively used. As mentioned in Section 8.5.3, we cannot compare the known implementation of promoters and inhibitors in Maude [AC09], since they use a different *dynamic* interpretation of this feature. We have also tried to compare the prototype with the kPWORKBENCH tool [GKI+15] based on kernel P systems. However, the graph-like structure of these systems is not directly translatable to the nested structure of those used in this chapter, its simulator randomly chooses an alternative for each evolution step while our command considers all of them, and the concepts of model used for model checking apparently do not coincide.

[7] Figure 8.3 shows as `check` the execution time and memory usage of the `membranes.py` script using the Python-based builtin model checker (see Section 4.4). Since the measure of the builtin backend does not include the interface initialization time, their results for reduced sizes are notably smaller.

Chapter 9

Population protocols and chemical reaction networks

Population protocols [AAD+06] and chemical reaction networks [Bri19] are two models of distributed computation, which can be compared at some extent to the membrane systems in the previous chapter. Population protocols are inspired on networks of mobile sensors that interact when they meet, while chemical reaction networks are based on chemical reactions in a closed system. Quantitative properties like convergence time are relevant for these computational models and they are usually studied in a probabilistic setting. In separate sections, the two formalisms are described, and how to represent them in Maude is discussed through examples. The probabilistic extensions described in Chapter 5 are essential to specify the probabilistic information they contain, and to verify and estimate quantitative properties about them.

Like Chapter 5, this chapter describes recent work that has been done during a research stay in the Chair of Foundations of Software Reliability and Theoretical Computer Science led by Javier Esparza at the Technische Universität München.

9.1 Population protocols

Population protocols [AAD+06] are a distributed computational model that involves a swarm of mobile agents with highly limited resources interacting with each other. Initially, they were introduced to study computation in networks of mobile sensors, but the formalism has been applied to other fields like robotics and chemistry. Agents are indistinguishable except for their current state, which is updated by pairwise interactions according to a common transition function. The number of states is typically small and they are associated to a binary vote that is used to decide predicates by consensus. Since agents are not aware of the global state of the computation, there is no notion of completion or termination.

Definition 9.1. *A* population protocol *is a tuple $P = (S, T, I, O)$ where S is a finite set of states, $T \subseteq S^2 \to S^2$ is a finite set of transition rules, $I \subseteq S$ is a subset of initial states, and $O : S \to \{0, 1\}$ maps each state to an output.*

A *configuration* is a multiset $C : S \to \mathbb{N}$ of agents, often seen as a vector $\mathbb{N}^{|S|}$. The *output* of a configuration is $O(C) = b$ if there is a $b \in \{0, 1\}$ such that $O(s) = b$ for all $s \in S$ with $C(s) > 0$, and otherwise is undefined $O(s) = \perp$. There is a transition between two configurations $C \to D$ if the number of agents in D is updated from C by a transition rule. A configuration C is *stable* if $O(C) \in \{0, 1\}$ and $O(D) = O(C)$ holds for every reachable configuration $C \to^* D$. An execution $C_0 \to C_1 \to C_2 \to \cdots$ is *fair* if all states that are reachable infinitely often are visited infinitely often, i.e. if $\{i \in \mathbb{N} : C_i = D\}$ is infinite whenever $\{i \in \mathbb{N} : C_i \to^* D\}$ is infinite.

Definition 9.2. *A protocol P computes a predicate $\varphi : (I \to \mathbb{N}) \to \{0, 1\}$ if for every initial configuration C any of the following equivalent conditions holds*

1. *every fair execution contains a stable configuration D such that $O(D) = \varphi(C)$, or*
2. *assuming that at each step of an execution two agents are picked uniformly at random for interacting, a stable configuration such that $O(D) = \varphi(C)$ is visited with probability one.*

It is known that the expressive power of population protocols is that of Presburger predicates or semilinear sets [AAE06]. A predicate is Presburger-definable if it can be defined in first-order logic over $\mathbb{N}$ with addition and order, or equivalently if it is a Boolean combination of *threshold* and *modulo* predicates of the form $\sum_{k=1}^{n} a_k x_k \leq b$ and $\sum_{k=1}^{n} a_k x_k \equiv b \pmod{c}$.

The work [BEJ+18] is a nice presentation of population protocols, the problems and questions concerning them, and the related verification techniques. There, population protocols are explained with the example of a group of black ninjas meeting at night in a Zen garden to decide whether they will attack some position of the enemy at dawn. The decision must be taken by majority and all ninjas must be aware of the decision, since everyone is required for the attack to be successful. Communication is only possible by direct contact between two agents, so the ninjas keep roaming around the garden to share their votes and calculate the final decision. However, they cannot identify the other ninjas they have interacted with nor memorize their votes, so their sensei should be creative to find out a procedure so that the decision can be successfully transmitted to all ninjas. It is not only important that the consensus is eventually reached, but also that this is achieved soon enough and namely before dawn.

In the following, we discuss the representation in Maude of the two simplest protocols described in that paper, as a model for specifying any other population protocol. Some more examples have been specified in Maude and are available in the example collection, including the third refinement of the ninjas' protocol we are omitting here. Agents and configurations are represented as terms and transitions as rewrite rules, in two alternative ways. Strategies are not needed at this moment, but we will use them to select some rules and disable others for experimentation. One high-level and flexible way of describing the configurations of the protocol is by means of a commutative and associative operator acting as a multiset of states. Every element of this multiset is an agent that cannot be distinguished from another agent in the same state.

```
mod NINJAS is
  protecting NAT .

  sorts Mode Vote Ninja Garden .
  subsort Ninja < Garden .

  ops A P : -> Mode [ctor] .
  ops Y N : -> Vote [ctor] .

  op {_,_} : Mode Vote -> Ninja [ctor] .

  op empty : -> Garden [ctor] .
  op __    : Garden Garden -> Garden [ctor assoc comm id: empty] .

  vars V W : Vote .
  vars K L : Nat .

  rl [ay&an] : {A, Y} {A, N} => {P, N} {P, N} .
  crl [a&p]  : {A, V} {P, W} => {A, V} {P, V} if V =/= W .
  rl [py&pn] : {P, Y} {P, N} => {P, N} {P, N} .
  op initial : Nat Nat -> Garden .
  eq initial(0, 0) = empty .
  eq initial(0, s K) = {A, N} initial(0, K) .
  eq initial(s K, L) = {A, Y} initial(K, L) .
endm
```

In the ninjas' protocols, an agent or a state is a pair $\{m,\ v\}$ where m is a mode, either Active or Passive, and v is a vote on the attack, either Yes or Not. Transitions are specified as rewrite rules that pick two agents of the configuration and update their states as follows,

1. (ay&an) If both agents are active and their votes are distinct, they become passive and the Y-voter changes its vote to N. Like this, distinct votes will be canceled pairwise and only agents for the majority vote will remain active. Moreover, in case of tie, the negative decision will prevail.

2. (a&p) If one agent is active and the other is passive, the passive one copies the vote of the active one.

3. (py&pn) If both agents are passive and their votes differ, as in the active case, the Y-voter becomes an N-voter.

The initial proposal of the sensei is using only the first and second rules, but it is discovered later that adding the third rule is convenient, although not as convenient as desired. We select them with the strategies sensei and senseii respectively.

```
smod NINJAS-STRAT is
  protecting NINJAS .

  strats sensei sensei-bis senseii @ Garden .

  sd sensei     := (ay&an | a&p) ! .
  sd sensei-bis := (ay&an or-else a&p) ! .
  sd senseii    := (ay&an | a&p | py&pn) ! .
endsm
```

All agents are active in the initial configuration, which is constructed by the initial(n, m) function with n votes for Y and m votes for N.

Formally, the set of states of the protocol is $S = T_{\Sigma,\texttt{Ninja}}$, the transitions are all the ground instances of the rules in the NINJAS module, the initial states are the active Ninja terms in which the first entry is A, and the output function O can be defined as $O(\{m,\ \texttt{Y}\}) = 1$ and $O(\{m,\ \texttt{N}\}) = 0$ for any mode m. These protocols are intended to compute the predicate $\varphi(C) = C(\{\texttt{A},\ \texttt{Y}\}) > C(\{\texttt{A},\ \texttt{N}\})$, i.e. to decide whether the initial number of voters for Y is greater than the initial number of voters for N. Remember that in the initial configuration, generated by initial, only active agents are present.

A population protocol is correct with respect to a predicate if and only if the conditions of Definition 9.2 are satisfied for every possible initial state. Even though initial states are infinitely many, verifying the correctness of a protocol is decidable by reducing it to the Petri net reachability problem [EGL+17], whose complexity is however hyper-Ackermaniann. The automated tool Peregrine [EHJ+20] is able to fully verify population protocols using stage graphs [BEJ+17] and an SMT solver. Their manual verification using theorem provers has also been tried [DM09]. Model checking as in Sections 2.3 and 2.7 is not suitable however to fully verify population protocols, since only instances of the protocol for a given initial configuration can be checked. Nevertheless, model checking has been used to gain confidence about the correctness of the protocols by checking some fixed instances of them [CDF+11]. According to the equivalent conditions in Definition 9.2, either qualitative or quantitative model checkers could be applied for those checks, although specifying whether an execution is fair in the sense of this definition is not directly feasible within a temporal property.

Let us come back to the ninjas' protocol and introduce two atomic propositions `consensus(Y)` and `consensus(N)` to signal the configurations where consensus for one or the other vote has been reached.

```
mod NINJAS-PRED is
  protecting NINJAS .
  including SATISFACTION .

  subsort Garden < State .

  op consensus : Vote → Prop [ctor] .

  var M : Mode .    var V : Vote .   var G : Garden .

  eq {M, N} G ⊨ consensus(Y) = false .
  eq {M, Y} G ⊨ consensus(N) = false .
  eq G        ⊨ consensus(V) = true [owise] .
endm
```

Stable consensus can be expressed in LTL by $\Diamond\Box\, consensus(v)$ for the vote v, but this property is only required for fair executions. Remember that fair executions are those in which the configurations that are reachable infinitely often are visited infinitely often. The direct translation of this statement would require using CTL* and identifying every state with an atomic proposition, but a weaker premise could be enough to prove the property. This is the case for the `sensei` protocol and the following initial configuration with 4 positive votes and 2 negatives votes.

```
$ umaudemc check ninjas 'initial(4, 2)' \
    'A (([] E <> consensus(Y) -> [] <> consensus(Y)) -> <> [] consensus(Y))' \
    sensei
The property is satisfied in the initial state
(9 system states, 107 rewrites, holds in 9/9 states)
```

Fixed a path with the **A** operator, the conclusion tells that a stable consensus on `Y` is reached. Its premise $\Box\,\mathbf{E}\Diamond\, consensus(\texttt{Y}) \to \Box\Diamond\, consensus(\texttt{Y})$ filters those executions that are fair with regard to the satisfaction of the $consensus(\texttt{Y})$ property. Since this is weaker than being a fair execution, the positive answer of the model checker ensures that the protocol is correct for this input, and the same happens for the `senseii` protocol. However, this does not mean that the protocols are correct, as they may fail on other inputs.

For an initial tie configuration like `initial(4, 4)`, the protocols are required to attain a negative consensus, but this does not actually happen with `sensei`. Checking the previous formula is not a way of refuting the correctness of the protocol, since it is stronger than the actual requirement. However, a witness of its failure can be obtained with the `srewrite` command.

```
Maude> srew [2] initial(4, 4) using sensei .

Solution 1
rewrites: 32
result Garden: {P,N} {P,N} {P,N} {P,N} {P,N} {P,N} {P,N} {P,N}

Solution 2
rewrites: 38
result Garden: {P,Y} {P,N} {P,N} {P,N} {P,N} {P,N} {P,N} {P,N}
```

The solutions of the `srewrite` command for the `sensei` strategy are the irreducible terms under the transition rules of the first protocol. An execution that reaches the second solution of the command and stays there forever is necessarily fair and a stable consensus is not reached in it, so the protocol is not correct. Adding the rule py&pn as in the senseii strategy solves the problem in this second solution, the presence of a positive vote that cannot be convinced by the other passive voters.

```
$ umaudemc check ninjas 'initial(4, 4)' \
    'A (([] E <> consensus(N) -> [] <> consensus(N)) -> <> [] consensus(N))' \
    senseii
The property is satisfied in the initial state
(23 system states, 297 rewrites, holds in 23/23 states)
```

The strategy sensei-bis in NINJAS-STRAT also solves the problem but in a way that destroys the distributed nature of the protocol, since it forces active agents to interact among them before doing so with passive ones.

Using the probabilistic interpretation of population protocols, their correctness for some fixed instances can be more easily checked. Moreover, quantitative properties like convergence times can be calculated. This can be done with the umaudemc's pcheck command introduced in Section 5.1 by adding probabilities to the model we have just used for the classical model checking. However, we will change to an alternative, more efficient but lower-level representation of the state.[1] Instead of a multiset of agents, we maintain a tuple with the number of agents for each state as in the formal definition of configuration.

```
mod NINJAS is
  protecting NAT .
  protecting QID .

  sort Garden .

  op <_,_,_,_> : Nat Nat Nat Nat -> Garden [ctor] .

  vars AY AN PY PN : Nat .

  rl [ay&an] : < s AY, s AN,   PY,   PN > => <   AY,   AN,   PY, s s PN > .
  rl [ay&pn] : < s AY,   AN,   PY, s PN > => < s AY,   AN, s PY,     PN > .
  rl [an&py] : <   AY, s AN, s PY,   PN > => <   AY, s AN,   PY,   s PN > .
  rl [py&pn] : <   AY,   AN, s PY, s PN > => <   AY,   AN,   PY, s s PN > .

  op weight : Garden Qid -> Nat .
  eq weight(< AY, AN, PY, PN >, 'ay&an) = AY * AN .
  eq weight(< AY, AN, PY, PN >, 'ay&pn) = AY * PN .
  eq weight(< AY, AN, PY, PN >, 'an&py) = AN * PY .
  eq weight(< AY, AN, PY, PN >, 'py&pn) = PY * PN .
endm
```

The rules have been adapted to the new tuple representation, and since the mode and the vote can no longer be handled independently, a&p has been split into ay&pn and an&py. Nevertheless, this representation facilitates the specification of the probabilities of the model, which is done by the weight function. Given the starting configuration of a transition and the label of the rule that has been applied, weight calculates the number of combinations of agents that can interact with this rule. Once normalized, these are the probabilities of selecting the two required agents uniformly at random, disregarding the combinations where no rule is applicable. This is what happens when pcheck is invoked with the term method to assign probabilities.

```
$ umaudemc pcheck ninjas-tuple '< 4, 4, 0, 0 >' '<> [] consensus(N)' \
```

[1]Maude internally represents associative and commutative operators with identity (ACU) in the most common situation as vectors mapping their arguments to their multiplicities. Our tuple-based representation of configurations is essentially the same but exposed to the user, so that we are allowed to obtain the count of agents in a state.

```
    sensei --assign 'term(weight(L:Garden, A:Qid))'
Result: 0.06713190566234181
$ umaudemc pcheck ninjas-tuple '< 4, 4, 0, 0 >' 'P ≥ 1 <> [] consensus(N)' \
    senseii --assign 'term(weight(L:Garden, A:Qid))'
The property is satisfied (23 system states, 78 rewrites)
```

The first result tells that sensei is not a correct protocol because stable consensus is not almost sure, and the second says that the senseii protocol is correct at least for this input. We have used the PCTL operator $\mathbf{P}_{\geq 1}$ in this case. Moreover, we can calculate and compare the expected time for reaching a consensus in the two protocols, using the --steps option of pcheck.

```
$ umaudemc pcheck ninjas-tuple '< 4, 3, 0, 0 >' '<> consensus(Y)' sensei \
    --assign 'term(weight(L:Garden, A:Qid))' --steps
Result: 11.181798343404381 (relative error 5.929972206885414e-06)
$ umaudemc pcheck ninjas-tuple '< 4, 3, 0, 0 >' '<> consensus(Y)' senseii \
    --assign 'term(weight(L:Garden, A:Qid))' --steps
Result: 825.9048662846063 (relative error 9.998262507239355e-06)
```

For this initial configuration in the border of a tie, the incorrect protocol sensei is quite faster than senseii, and the same happens for other configurations when they both compute the correct consensus. Another refinement of these protocols in [BEJ+18] solves the slow speed of senseii.

Instead of the pcheck command, we could have used the probabilistic extension of the strategy language to simulate the population protocols. On top of the tuple-based specification, an iteration of the protocol can be specified by the following step strategy that applies one of the rules of the senseii protocol with the same weights we have indicated in the weight function.

```
  sd step := matchrew G s.t. < AY, AN, PY, PN > := G by G using choice(
                AY * AN : ay&an,
                AY * PN : ay&pn,
                AN * PY : an&py,
                PY * PN : py&pn,
       ) .
```

We also define the following repeat strategy to execute a given number of iterations:

```
  sd repeat(0)   := idle .
  sd repeat(s N) := step ? repeat(N) : idle .
```

The execution stops either when the iteration count is exhausted or when an irreducible term by the step strategy is found. By running multiple simulations, we can see that the consensus is not attained with probability one within 20 steps, but it is when 100 steps are executed. However, the probability is already greater than 0.9 after 35 steps. The expected value calculated by probabilistic model checking with the pcheck command for this configuration is 18.06.

```
$ simaude ninjas-choice.maude '< 4, 2, 0, 0 >' 'repeat(20)'
Simulation finished after 100 simulations and 7612 rewrites.

  < 2,0,0,4 > 0.07
  < 2,0,2,2 > 0.27
  < 2,0,4,0 > 0.66

$ simaude ninjas-choice.maude '< 4, 2, 0, 0 >' 'repeat(100)'
Simulation finished after 100 simulations and 41976 rewrites.

  < 2,0,4,0 > 1
```

To conclude this section, we present a third representation of the senseii protocol based on its continuous-time semantics, in which each agent is endowed with an exponential clock of unitary period that activates it to interact with another agent selected at random. However, this is equivalent to having a single clock ticking at period $1/n$ and selecting two agents at random as in the previous specifications. Hence, we can adapt them without much effort to include the time variable. A Float argument is added to the Garden tuple along with a new rule to tick the clock at an exponential rate.

```
op <_,_,_,_,_> : Nat Nat Nat Nat Float → Garden [ctor] .

vars T DT : Float .

rl [tick] : < AY, AN, PY, PN, T > ⇒ < AY, AN, PY, PN, T + DT > [nonexec] .
```

Using the sample strategy with an exponential distribution and instantiating the DT variable of the tick rule with the sampled value, the time will advance as required. An execution of the continuous-time protocol for a given number of steps is implemented by the following repeat strategy.

```
sd repeat(R, 0)    := idle .
sd repeat(R, s(N)) := (sample DT := exp(R) in tick[DT ← DT]) ;
                      step ? idle : repeat(R, N) .
```

The rate R is set to the size n of the population. Simulating the new repeat strategy with the simaude tool, we obtain many solutions with different times, but always with the same configuration if the number of steps is large enough.

```
$ simaude ninjas-ct.maude '< 4, 2, 0, 0, 0.0 >' 'repeat(6.0, 100)'
Simulation finished after 100 simulations and 12364 rewrites.

  < 2,0,4,0,1.6774102257284829 >      0.01
  ...
  < 2,0,4,0,7.0349806876337002 >      0.01
  ...
  < 2,0,4,0,1.6283429704028018 >      0.01
```

Although it is not necessary in general, this particular protocol stops when a consensus is reached, so the weighted mean of the time values above estimates the expected convergence time of the protocol, 2.98 seconds. Notice that the product of this number and the rate of the global clock approximately gives 18, which is the expected number of steps obtained for the discrete-time setting. This is a general property, so adding the time explicitly in the specification is not much interesting.

Since the choice operators we have written in this section and the weights we have passed to the term assignment method are a direct consequence of the rules, they can be generated automatically by a reflective transformation within Maude. Obtaining Maude specifications from population protocols described in other languages is also straightforward. These transformations have been done for the chemical reaction networks explained in the next section.

9.2 Chemical reaction networks

Chemical reaction networks (CRN) [Bri19] are a simplified model of the behavior of chemical reactions, which have been used for the theoretical study of chemistry, but also as an abstract computational model. Solutions are represented by multisets of abstract chemical species or molecules that react according to a finite set of multiset rules. These reactions are similar to the evolution rules of membrane systems in Chapter 8, but they do not include communication messages and they are applied differently. Chemical reaction networks can be interpreted in a discrete sense or in a probabilistic sense as continuous-time Markov chains, and they are almost equivalent to population protocols when reactions are bimolecular.

Definition 9.3. *A* chemical reaction network *(CRN) is a pair* (S, R) *where* S *is a finite set of* species *and* $R \subseteq (S \to \mathbb{N}) \times (S \to \mathbb{N}) \times \mathbb{R}$ *a finite set of* reactions *over* A.

A reaction $\alpha = (r, p, k) \in R$ consists of two multisets specifying the reactants r they consume and the products p they generate (its stoichiometry), and a rate $k > 0$ expressing how likely is the reaction to occur. The *states* of a CRN are multisets of species $c : S \to \mathbb{N}$, and a reaction is applicable to a configuration c if $c \geq r$ and the result is a state $c' = c - r + p$.[2] This transition is written as $c \to^{\alpha} c'$ or $c \to c'$ without specifying the reaction. Like in population protocols, votes can be assigned to the species to compute predicates with or without introducing probabilities. A *chemical reaction decider* is a CRN along with a subset $I \subseteq S$ of initial states and two disjoint sets $S_0, S_1 \subseteq S$ of 0 and 1 voters. The output of a state is $b \in \{0, 1\}$ if $c(s) > 0$ for some $s \in S_b$ and $c(s) = 0$ for all $s \in S_{(1-b)}$. This can be used to define how CRNs compute predicates by reaching stable states as in population protocols, and their expressive power is actually the same. However, in the probabilistic setting and considering stability in the limit, they have been proven to compute predicates in the Δ_2^0 class of the arithmetical hierarchy with probability one, which is a superset of the Turing-computable predicates [CDS16].

From a probabilistic point of view, chemical reaction networks can be seen as continuous-time Markov chains (defined in Section 2.7). The *propensity* $\rho(c, \alpha)$ of a reaction $\alpha = (r, p, k)$ in a configuration c is $k \cdot \prod_{s \in S} C_{c(s), r(s)}$ where $C_{n,k}$ is the number of combinations of k elements out of n objects without replacement. Since $C_{n,k} = 0$ if $n < k$, reactions that are not applicable get a null propensity. In other words, the propensity of a reaction is the product of its rate by the number of matches of its reactants in the multiset configuration. The total propensity at c is $\rho(c) = \sum_{\alpha \in R} \rho(c, \alpha)$. Then, the semantics of a CRN (S, R) can be defined by the continuous-time Markov chain $(S \to \mathbb{N}, T, P_0)$ with

$$T(c, c') = \textstyle\sum_{c \to^{\alpha} c'} \rho(c, \alpha)$$

The initial distribution P_0 can be defined for a given initial state as $P_0(s_0) = 1$ and $P_0(s) = 0$ for all $s \neq s_0$. This means that the time for passing from c to c' is governed by an exponential distribution whose rate is the sum of the propensities of the reactions that produce that change. In other words, as we mentioned in Section 2.7, the expected time until the next reaction occurs is $\rho(c) = \sum_{c' \in S} T(c, c')$ and the probability of applying a particular reaction α is $\rho(\alpha, c) / \rho(c)$. Regardless of time, CRNs can also be seen as discrete-time Markov chains with the aforementioned probabilities.

As we did for population protocols, we aim to show how to specify chemical reaction networks in Maude using an example. In fact, the representations of population protocols and CRNs are essentially the same and the only novelty is time, on which we have already discussed for the continuous-time semantics of the former. The multiset of species in chemical reaction networks can be represented as an associative and commutative operator too, but we will directly use the tuple-based encoding we have already introduced for population protocols to avoid being reiterative. The following module `PRED-PREY` specifies an ecosystem where two biological species, say wolves and rabbits, interact as predators and preys. The population of preys grows independently by the `prey` reaction as they reproduce, the number of predators naturally decreases by the `pred` reaction as they die, and their population only augments when they prey with the `prey_pred` reaction. This model is a discretization of the well-known Lotka-Volterra differential equations, which have been originated in the chemistry field [Lok10] despite its popularization for biological systems.

```
mod PRED-PREY is
  protecting NAT .
```

[2]Abusing notation, we write $f \; R \; g = \bigwedge_{s \in S} f(s) \; R \; g(s)$ for any relation R, and $(f \otimes g)(s) = f(s) \otimes g(s)$ for any binary function $\otimes$.

```
  protecting FLOAT .

  sort System .

  op <_,_,_> : Nat Nat Float → System [ctor] .

  vars Prey Pred : Nat .
  vars T DT : Float .

  rl [prey]      : <   s Prey,     Pred, T      >
                ⇒ < s s Prey,     Pred, T + DT > [nonexec metadata "10.0"] .
  rl [prey_pred] : <   s Prey,   s Pred, T      >
                ⇒ <     Prey, s s Pred, T + DT > [nonexec metadata "0.01"] .
  rl [pred]      : <     Prey,   s Pred, T      >
                ⇒ <     Prey,     Pred, T + DT > [nonexec metadata "10.0"] .

  op initial : → System .
  eq initial = < 1000, 1000, 0.0 > .
endm
```

The state of sort `System` is a tuple with the number of molecules of each species and the reaction time. The rules `prey`, `prey_pred`, and `pred` of the CRN include a free variable `DT` that increases the time variable. Alternatively, we could have implemented the increment of time as a separate rule, so that reactions are still executable without strategies. In the `metadata` attribute of the rules, only for informative purposes, we have written their rates. The reaction `prey_pred` is 10^4 times slower than the other two reactions. In this CRN, the number of species is unbounded because of the `prey` rule, so we can only run stochastic simulations on it. Reactions should be selected according to their propensities, for what the **choice** operator introduced in Section 5.2 is needed.

```
smod PRED-PREY-STRAT is
  protecting PRED-PREY .
  protecting CONVERSION .

  strat step @ System .

  var  S         : System .
  vars Prey Pred : Nat .
  vars T DT      : Float .

  sd step := matchrew S s.t. < Prey, Pred, T > := S
             /\ DT := 1.0 / (float(10 * Prey) + 0.01 * float(Prey * Pred)
                             + float(10 * Pred))
           by S using choice(
                10 * Prey                  : prey[DT ← DT],
                0.01 * float(Prey * Pred) : prey_pred[DT ← DT],
                10 * Pred                  : pred[DT ← DT]
           ) .
endsm
```

The weight of each reaction α in the **choice** combinator is the propensity $\rho(c, \alpha)$, where the counts of species in c are obtained by matching with **matchrew**. The elapsed time `DT` is calculated as the inverse of the total propensity $\rho(c)$, which coincides with the mean period of the global clock, because the mean of an exponential distribution is the inverse of its rate. Since this value is used to instantiate the homonym variables of the reaction rules, the accumulated time

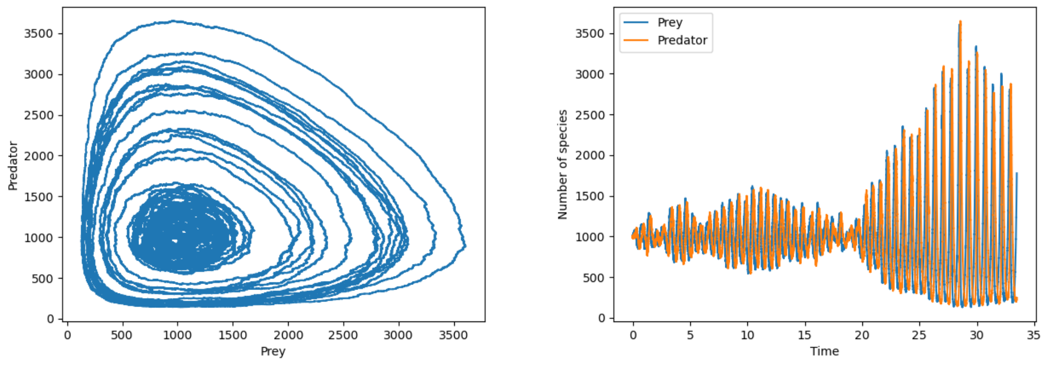


Figure 9.1: Evolution of the CRN in `PRED-PREY` in a simulation of 10^6 steps.

stored in the state is increased by this amount. Strictly, we should have sampled an exponential random variable with rate $\rho(c)$ to obtain the time increment `DT`, and this does not pose any difficulty.

```
sd step := matchrew S s.t. < Prey, Pred, T > := S by S using
  sample DT := exp((float(10 * Prey) + ... + float(10 * Pred))) in choice(...) .
```

However, time is an output-only variable that does not condition the evolution of the network, and so we can take the means of the distributions instead of sampling them to save work. In effect, the expected value of the sum of this sequence of samples will be the sum of their means. After simulating a million of consecutive executions of the `step` strategy with a total elapsed time of 33.47 units, Figure 9.1 shows the evolution of the number of predators and preys from the initial state `initial`. The left curve is the trajectory described by the number of preys and the number of predators, and the right plot represents both quantities with respect to time. While the solutions of the continuous Lokta-Volterra equations are closed periodic curves similar to what we have obtained in the left plot, the discretization allows escaping from them with certain probability and reaching another orbit. These figures have been generated by evaluating the `step` strategy repeatedly on the term produced in the previous step using the `maude` Python library (see Appendix A).

This manual specification of chemical reaction networks can be partially automated by inferring the propensities in the **`choice`** operator from the rules, if they are annotated with their rates in the `metadata` attribute like in the `PRED-PREY` module. Composing the `step` strategy is straightforward once we have generated the propensity expressions of the **`choice`** operator by looking at the rates and the number of successors in the lefthand sides of the rules. Generating Maude specifications of this form from other representations of chemical reaction networks is not difficult too, and we have written a template-based translator in Python from networks specified in the format of the SeQuaiA [CCK20] tool. This software performs semiquantitative analysis of CRNs by computing and operating on abstractions where the state space has been partitioned. We have used the generated Maude specification to run the bounded simulations required to compute these abstractions, but the performance is substantially worse than the other implementations in different imperative programming languages. This is not surprising, since Maude is not designed for numerical simulations, its natural numbers in the `Nat` sort are arbitrary precision integers not directly supported by the hardware, and there is no random access to data structures. Moreover, efficient methods for simulating chemical reaction networks and population protocols [BHK+20] *accelerate* the simulations by executing multiple steps at once, and this could not be efficiently done with a strategy in Maude unless the original rules are replaced by generalizations like the following where the parameter `N` is the number of accelerated steps.

```
crl [prey_pred] : < Prey, Pred, T > ⇒ < Prey - N, Pred + N, T >
 if Prey ⩾ N /\ Pred ⩾ N [nonexec metadata "0.01"] .
```

However, the flexibility of Maude can still be useful to execute experiments and easily change between different options.

Chapter 10

Performance evaluation

In this section, we evaluate the performance of the C++ implementation of the Maude strategy language presented in Chapter 3 with respect to the original Maude prototype, of the strategy-aware extension of the Maude LTL model checker explained in Section 4.3 compared with the original one and Spin, and among the different connections with external model checkers described in Section 4.4. The specifications used in the comparisons, some of them described in this thesis, are available in the example collection repository, where instructions are included for the reproducibility of the results. All executions in the following sections have been carried out on Linux in multiuser non-graphical mode in an Intel Core i7 8550U with 11.6 Gb of RAM.

10.1 Integration of the strategy language in Core Maude

As explained in previous chapters, the strategy language was first implemented in Maude as an extension of Full Maude. We have compared the performance of the new C++ implementation with respect to that prototype using the original examples written for it, which are summarized in Chapter 6. Since we have fixed some bugs and made some improvements when updating these examples to the current version of the language, instead of using their original code for the benchmarks, we have slightly adapted the renewed versions to run in the prototype. Additionally, we have included the 15-puzzle in the comparison, since it is the running example of the strategy language chapter. Moreover, the strategy language prototype was based on an old version of Full Maude that is incompatible with the current Maude releases, so we have minimally adapted it to run in Full Maude 3.1. This version of the extended interpreter comes with the strategy language as standard through the Core Maude implementation, but we have disabled what may interfere with the prototype. Notice that the meaning of the strategy search command differs from the prototype to the current version: the former's `srew` command looks for a single solution of the strategy, and we identify it with the new `dsrew [1]` command; its `srewall` command is the current `srew` that looks for all solutions of the strategy.

Table 10.1 collects some measures of the execution of both implementations on each example. The last three columns show the quotient of different measures with the C++ implementation in the denominator, so that the improvement is greater as these numbers are. The total time and number of rewrites are calculated as the sum of those printed by Maude for each command, thus excluding module parsing time. The value of the memory peak refers to the whole example execution, and the memory used by the interpreter and the prototype have been subtracted before calculating the coefficients. Otherwise, the prototype always uses much more memory due to Full Maude.

While improvement is not uniform in all examples, unsurprisingly, the C++ outperforms the Maude-based one in all of them. For the semantics of the ambient calculus, which is the example with the highest absolute execution time, the speedup is very significant. The memory usage is also drastically reduced. In the specification of Eden, a parallel Haskell-like

	Time (ms)		Measure proportion (old/new)		
Example	New	Old	Time	Rewrites	Memory peak
Sudoku [SP07]	908	90639	99.82	19.9	20.39
Neural networks [SPV09]	1061	26496	24.97	8.16	1.64
Eden [HVO07]	12	1360	113.33	417.61	2.74
Ambient calculus [RSV06]	16	125735	7858.44	85759.7	180.47
Basic completion (Section 6.5)	91	7322	80.46	66.56	2.2
Blackboard [MMV04]	1373	47027	34.25	5451.67	0.27
15 puzzle (Section 6.6)	182	2232	12.26	4.69	3.02

Table 10.1: Performance comparison between the original prototype and the C++ implementation.

programming language, the difference is not bigger because we have not included examples that finish in few milliseconds in the C++ implementation but take an excessive amount of time in the prototype. The same happens with the basic completion procedures of Section 6.5. The higher memory usage peak of C++ implementation in the blackboard example is caused by the allocation of nodes of the directed acyclic graph in which terms are represented, and it could be explained by the way Maude reserves and reuse memory for them. In absolute terms, the memory peak of the Core Maude implementation is 26.71 Mb while that of the prototype is 119.17 Mb, and the total memory even after discounting the total memory used by Full Maude alone is 22.77 times greater in the Maude-based implementation.

This comparison exhibits some other advantages of the new implementation in addition to performance, since translating some of these examples back to the old prototype was not an easy task in some cases. Its strategy modules do not admit parameterization, module importation, or even selecting which module is being controlled. Hence, modular and parametric strategy specifications must be flattened in a single module of the prototype. This is specially visible in the basic completion example. Moreover, the syntactical analysis of strategy modules is less robust in the prototype, errors are less informative, and strategies with more than two arguments cannot be declared.

10.2 Extension of the Maude LTL model checker

The extension of the Maude LTL model checker and its connection with external tools have been tested with several examples of temporal properties on strategy-controlled specifications available in the example collection of the Maude strategy language, as explained in Section 10.3. In order to evaluate the performance of the extension objectively, strategy-controlled models should be checked with this and other tools while measuring their executions. However, there are no other model checkers for strategy-based specifications, so we would have to specify the same problem in the potentially very different formalism of the other tool, which may not allow a faithful and easy translation. Since our tool is an extension of the Maude LTL model checker, measuring the executions of both model checkers for pairs of strategy-controlled and equivalent rule-only Maude specifications would be a targeted and significant comparison. Not only a great part of both specifications could be shared, but an important part of the model-checking algorithms is shared too, so the evaluation is focused on the extension itself and in its essential novelty, the generation of the system automaton. We have translated the strategy-based specification of the examples in Sections 6.2 and 6.3 to rule-based ones for each strategy, and the results are included in the following sections. In the first case, we have also specified the example in the Promela language and model checked it using the well-known Spin model checker [Hol⁺21].

Performance improvements cannot be expected in general as a consequence of using strate-

Example	States			Time (ms)			Memory (Mb)		
	Spin	This	Maude	Spin	This	Maude	Spin	This	Maude
`train`	67919	723	2512	45	20	24	133.91	9.07	11.27
`bakery`	1539	2617	1292	80	43	9	128.73	16.17	10.25
`zune`	263	118	50	73	1	1	128.73	8.2	8

Figure 10.1: Comparison of examples in the Spin distribution.

gies to specify systems, since they are used as a high-level control resource intended to write clearer specifications and experiment easier with them, and this additional abstraction may occasionally have some cost. On the contrary, when strategies are used to restrict the state space for the sole purpose of verification or to conduct the model checker towards conjectured counterexamples, efficiency is actually the goal and should be surely expected. However, we are not making such a use of strategies in this thesis, which is mentioned in Section 11.1. In the following examples, the changes in the data representation and the rules that have replaced the strategies are as or ever more efficient than the original strategies, but the specifications are more obscure and need to be distinct for each control mechanism. Similarly, by translating the first example specification to the lower-level language of the Spin model checker, the performance has improved at some cost in readability.

In the first example in Section 10.2.1, the results show that there is no significant performance improvement in translating strategies into rules, and that the differences are not uniform and may depend on the strategy. In effect, two strategies are considered in that case, and the execution of the first one takes more memory and time than the rule-only translation, while the translation of the second is even less efficient than the direct execution of the strategy. Worse results are obtained with the second example in Section 10.2.2, which deals with the worst case of the model checker implementation described in Section 4.7.

However, using strategies when writing specifications from scratch or adding strategies to existing implementations may help to build more efficient as well as more readable models. An example is the membrane system environment in Section 8.6. In this case, the strategy-based model checker is way more efficient than the previous prototype, probably because strategies allowed introducing improvements easily. Moreover, we have translated some small examples included in the distribution of the Spin model checker, and the better results are obtained within Maude. These examples were originally specified in Promela, the modeling language of Spin, and have been translated to Maude both using and not using strategies. All LTL formulae there included have been checked, except `c7` in `train`, and the results are in Figure 10.1. Most of these properties do not hold, so the number of states may depend on how lucky the exploration order of each tool has been. However, these figures have also improved due to the simpler specifications that can be easily written in Maude. For instance, Spin uses a dedicated process and multiple message queues to implement a queue for the train gate in the `train` example, while this can be easily and more efficiently handled with a list in Maude. Moreover, fixing some order restrictions that do not affect the satisfaction of the properties is very natural with strategies. The total verification time has been calculated by adding the `modelCheck` term evaluation times reported by Maude and the execution times of the verifier generated by Spin for each property. If we had counted the initialization of Maude and the generation of the verifier by Spin, the results would also benefit the first. The high memory usage in Spin is due to an internal hash table, whose size is configurable but we have kept in its default values.

In Section 10.3 more examples are checked with different model checkers, but all of them are strategy-controlled specifications executed by Maude and exported as low-level Kripke structures to the other tools.

Num	States	Time (ms)			Rewrites		Memory peak (Mb)		
phil	All	SL	Maude	Spin	SL	Maude	SL	Maude	Spin
3	12	37	37	1161	118	99	8.17	8.14	128.8
5	48	38	37	1187	548	493	8.21	8.16	128.8
7	180	43	39	1235	2354	2191	8.58	8.48	128.8
11	2268	160	68	1286	36962	35503	11.76	10.39	128.8
13	7776	537	168	1329	139316	134941	20.65	14.44	128.8
17	87480	8876	2216	1357	1.87e6	1.83e6	178.25	107.22	128.8
23	3.07e6	-	-	4029	-	-	-	-	550.4
27	3.19e7	-	-	40299	-	-	-	-	5241.8

(a) Someone eats $\Diamond \bigvee_{k=0}^{n-1} \texttt{eats}(k)$ with `parity`

Num	States			Time (ms)			Rewrites	
phil	SL	M	Spin	SL	Maude	Spin	SL	Maude
3	10	9	66	38	37	1160	137	137
5	16	15	170	38	38	1187	541	553
7	22	21	332	41	40	1235	2077	2109
11	34	33	770	89	89	3987	31981	32077
13	40	39	1066	245	245	27596	1.27e5	1.27e5
17	52	51	-	3387	3418	-	2.03e6	2.03e6
23	70	69	-	234431	229617	-	1.3e8	1.3e8

(b) All eat $\Box \bigwedge_{k=0}^{n-1} \Diamond \texttt{eats}(k)$ with `turns`

Table 10.2: Execution measures for the philosophers problem using Maude and Spin.

10.2.1 The philosophers problem

As we mentioned in Section 6.2, the dining philosophers problem can be generalized to n philosophers and n forks without modifying its terms, rules, and strategies. Only the initial term and the temporal formulae have to be adapted, but they are defined so that the number of philosophers is received as a parameter. Table 10.2 shows under the SL columns the number of states, the time in milliseconds, the number of rewrites, and the peak usage of heap memory spent to model check the two considered LTL properties in the strategy-controlled specification with an increasing number of philosophers. As a reference, the number of states in the uncontrolled system is 3^n. The last row for the `parity` strategy is empty since the model checker does not finish in reasonable time for that number of states. All measures grow exponentially as the number of states, including the amount of memory used for the first property, which reaches 2.12 Gb for $n = 21$ and becomes unfeasible for $n = 23$. On the contrary, the memory peak using the `turns` strategy stays low and stable.

We may inquire whether a better performance could be obtained if instead of specifying these restrictions as strategies we modify the system module so that rules incorporate them, albeit the other advantages of strategies would be lost. In the case of the `parity` strategy, the `left` and `right` rules are implemented by the following five rules:

```
crl [left-even]  : ψ (o | Id | o) ⇒ (ψ | Id | o) if 2 divides Id .
rl  [left-odd]   : ψ (o | Id | ψ) ⇒ (ψ | Id | ψ) .
crl [right-odd]  : (o | Id | o) ψ ⇒ (o | Id | ψ) if not(2 divides Id) .
rl  [right-even] : (ψ | Id | o) ψ ⇒ (ψ | Id | ψ) .
rl  [left-even]  : < (o | Id | o) L ψ > ⇒ < (ψ | Id | o) L > .
```

The turns strategy has also been implemented without strategies by using a token passed to the next philosopher within the rules. Under the Maude columns of Table 10.2, there are the results of checking the same properties using the standard model checker on the transformed specifications. In the parity case, the number of states does not change and the other measures are lower in the transformed system. However, the critical number in which verification is no longer feasible coincides (the modified system takes 1.20 Gb with $n = 21$). In the case of turns, the figures are equivalent or even better for the original specification. No more than 8.7 Mb of memory are used both with and without strategies. Hence, at least for this problem, there is no significant performance loss on using strategies. The greater usage of memory of the strategy-aware model checker can be explained by a second cache of the evaluation of atomic propositions on states in addition to that already provided by the common infrastructure. In general, although not in this case, different states of the strategy-controlled model may represent the same term, and this cache tries to avoid the evaluation of the same property not only on the same state, but on the same term. This feature can be disabled at compile time to reduce the memory consumption.

We have also specified this same problem in the Promela language of the Spin [Hol+21] model checker. The model consists of two byte arrays of length n describing the availability of each fork and the number of forks retained by each philosopher, which are updated by a process for each philosopher in a loop that implements the parity restriction or the turns strategy using an auxiliary variable for the current turn. The verification process in Spin consists of generating a C verifier from the Promela specification and the LTL formula using the `spin -a` command, compiling it with the C compiler, where we have used the `-O2` optimization flag, and running the resulting program. The measures of the execution of the last binary are included in Table 10.2, showing that its performance is noticeably better in the parity case. While both Maude specifications cannot handle in reasonable time and memory limits the size $n = 23$, Spin verifies this case in two seconds and can reach up to 27 philosophers with 12 Gb of RAM.[1] On the contrary, its behavior for the turns strategy is much worse. Once generated, the execution time of the verifier is small, but the first phase's time quickly grows due to the processing of the temporal formula. We have interrupted the `spin -a` command for $n = 17$ after ten minutes, while this case can be checked in less that 4 seconds in Maude.

10.2.2 Scheduling policies

The roundRobin strategy and its preemptive version in the example on scheduling policies in Section 6.3 have also been translated to rule-only Maude specifications, by extending the machine state.

```
op {_,_,_,_} : Soup Memory List{Pid} Mark → MachineState [ctor] .
op {_,_,_,_,_} : Soup Memory List{Pid} Nat Mark → MachineState [ctor] .
```

The list of process identifiers and the preemption counter maintained in the strategy arguments are stored in the machine state, which also includes a mark that will help to define the modified rules. Strategies allow using the failure of the execution of a process to switch to the next of the list, and this cannot be easily handled within the rules. In summary, we have solved the problem by modifying the rules where a process can get blocked to explicitly treat the negative case, switching to another process that can take a step.

```
crl [exec] : {[I, wait(Q) ; R] | [J, P] | S, M, I J PL, G}
          ⇒ {[I, wait(Q) ; R] | [J, P'] | S', M', PL', m(G)}
 if [Q, N] RM := M
 /\ N ≤ 0
```

[1]As we have mentioned before, the fixed value of 128.8 for the memory usage of Spin in the smaller cases is due to a hash table reserved by the model checker in its default setting, which we have not changed.

p	n	States		Time (ms)		Rewrites		Memory peak (Mb)	
		SL	Maude	SL	Maude	SL	Maude	SL	Maude
`pIo`	4	705	321	73	68	6201	3672	10.12	9.88
	6	28501	9781	505	214	457301	191176	20.6	15.34
	9	1.98e7	4.93e6	8.52e5	1.91e5	6.24e8	1.79e8	8765.37	3092.26
`p`	4	1621	825	99	84	37737	17568	10.73	10.17
	6	71107	24901	3163	762	4.47e6	1.10e6	52.8	23.37
	8	-	1.39e6	-	90263	-	1.11e8	-	828.56

Table 10.3: Execution measures for the scheduling policies example.

```
/\ pidsIn(S) subset list2set(PL)
/\ {[J, P] | S, M, J PL I, pending} ⇒ {[J, P'] | S', M', PL', done} .
```

One of the conditions is that all processes in the soup are in the list of processes, since otherwise the `roundRobin` strategy would try giving the processor to missing processes first, for what another rule is required. The mark at the end of the state is to ensure that one and only one step (the first one of the new active processes) is executed in the rewriting condition.

The results of the verification of the property $\Box\Diamond\, \mathit{inCrit}\,(1)$ for the initial states `initial(`n`,` p`)` for $p \in$ {`pIo`, `p`} using the `roundRobin` and its preemptive version respectively are shown in Table 10.3. These strategies fall in the worst cases of the implementation described in Section 3.7, where several **`matchrew`**s defer the detection of cycles, increasing the number of the model states. All measures decrease noticeably in the translated specification, but both become unmanageable for almost the same sizes.

Although model checking the strategy-controlled system provides a worse performance in this case, strategies are still useful for their greater flexibility. However, as future work, we should consider updating the implementation to improve how **`matchrew`**s are handled and its performance.

10.3 Connection to external model checkers

Using several strategy-controlled Maude specifications and temporal properties available in the example collection of the strategy language (see Section 1.2), we have tested and compared the performance of the model checkers supported by the `umaudemc` tool. The test suite includes 156 different cases, 130 of which are strategy-controlled, with properties in LTL (116), CTL (18), CTL* (4), and μ-calculus (14). We have measured the execution time, the number of states, the number of rewrites, and the length of the returned counterexamples. These results cannot be interpreted as a comparison of the external model checker themselves, because the figures are also affected by the efficiency of the connections to our Maude models through `umaudemc`. They are also affected by the generation of the strategy-controlled transition system, which may depend on the exploration order, and its adjustments for branching-time properties. Since the system automata are common to all of them, the results do not evaluate the model generation either, but they provide interesting information about it. The executions and measures have been carried out by `test` subcommand of `umaudemc` according to a test suite specification included in the example collection.

The plots in Figure 10.2 compare the time spent by the different backends to execute the same model-checking problems ordered by their number of states. Linear- and branching-time properties appear in the left and right figure respectively. For every test case c and backend b, the plots show a marker whose shape and color is determined by b in the coordinates

$$(n_c^{\text{LTSmin}}, (t_c^b - t_e^b) \,/\, (t_c^{\text{LTSmin}} - t_e^{\text{LTSmin}})),$$

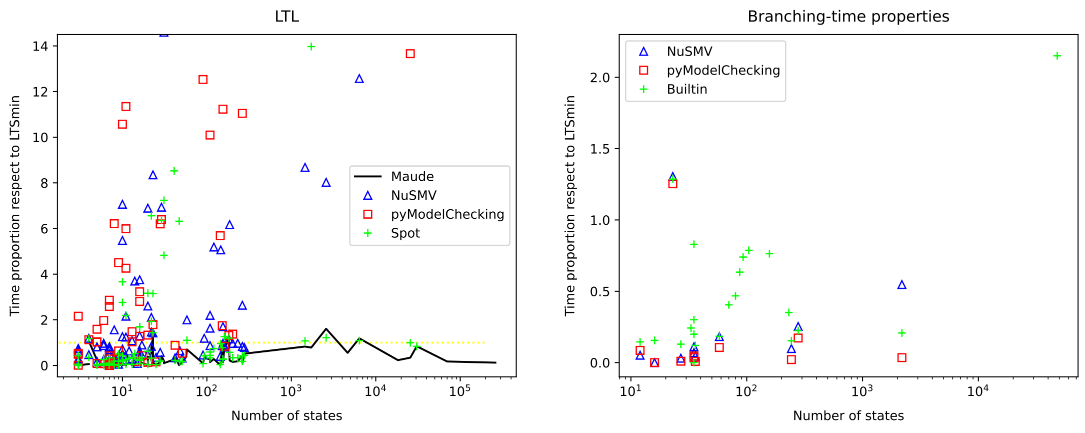


Figure 10.2: Compared performance of the model-checking backends.

where n_c^b and t_c^b are respectively the number of states and the execution time of the test case c in the backend b, and e is an empty test case with a single state and a trivial property. In other words, looking at a fixed vertical rule we can compare the performance of the backends for a test case, since the height of the marks indicates the proportion of time a backend has taken to complete its task with respect to LTSmin, with higher meaning worse. LTSmin has been chosen as a reference because it supports all considered logics. Since small examples are highly influenced by the quite different initialization times of the backends, where LTSmin is a thousand times slower on an empty example than the Maude LTL model checker, we have subtracted the initialization time before calculating the coefficients. Some marks lay out of the plot because their executions are much slower: 33 for `pyModelChecking` up to 11403 times slower, 7 for NuSMV up to 1382 times, and 4 for Spot up to 64 times. Moreover, `pyModelChecking` cannot even handle the 16.38% of the LTL test cases, and NuSMV the 7.76% of the LTL cases and the 11.11% of the CTL cases. The reasons are that either the program has run out of memory or we have interrupted the execution after a long delay.

For LTL properties, the Maude model checker is usually the fastest, albeit Spot and LTSmin are usually very close. The advantage of the builtin Maude model checker is not surprising, since it is directly connected to the rewrite graphs. Unexpectedly, the Python-based algorithms are more efficient than LTSmin, but only with small examples. The reason for the peak in the Maude curve is the order in which states are explored, despite they are true properties in which the whole state space has to be expanded, as we explain in the items at the end of the section. NuSMV and `pyModelChecking` do not behave badly for small examples, even though their algorithms do not operate on the fly. However, they do not terminate in reasonable time and memory limits when the problems are big enough.

Regarding branching-time properties, it is surprising that LTSmin exhibits a poor performance in most test cases, since its connection with the Maude models is more efficient and it is implemented in C++. This cannot be attributed to the longer initialization times of LTSmin, because they are subtracted before calculating the quotients. However, the problem sizes are small and its performance improves on bigger examples. Our own μ-calculus implementation, which also supports CTL formulae, is comparable to the other backends.

Looking at some additional information obtained from the execution of the test cases, the results and other details of the strategy-controlled model generation can be better explained:

- As we have mentioned in Section 4.7, the detection of equivalent states in the strategy-controlled model checker is sometimes delayed and sometimes incomplete, in a compromise between the reduction of the state space on the one hand and the memory and effort required to compute it on the other hand. The implementation outputs the number of total states and the number of real states after removing those that have been lately found equivalent. These numbers differ in the 17.14% of the cases using Maude and in the 15.75% using LTSmin. In this case, real states are the 76.57% and the 76.68% of all states respectively. When these numbers are equal, this does not necessarily mean that all equivalences have been found immediately, but there may be some that have not been detected at all. For example, there is a case in which the system automaton contains 25637 total states using both Maude and LTSmin, but only with LTSmin 5304 are found identical to some others. This does not affect the graph presented to the LTL model checker, but some work may have been saved in the execution of the strategies, which explains why the quotient between the Maude and LTSmin execution times is higher than the mean. The differences between the backends are explained by their different exploration orders, which may facilitate the detection of equivalence as states are created.

- The number of states also varies between some backends, not only on false properties because counterexamples could be found sooner or later, but also on true properties because of the incompleteness of the equivalence detection. No difference can be seen in the backends whose models are generated in `umaudemc`, since the state space is explored likewise. Maude expands fewer states than LTSmin in the 33.66% of the test cases, and in the 15.73% for true properties. The opposite situation happens in 10.89% and 6.25% of the cases, and the differences in favor of LTSmin are lower.

- For all external backends except LTSmin, model checking is executed in two phases, a preparatory one in which the system model is generated by `umaudemc` as required by the target tool, and the execution of the external model checker itself. The first phase supposes the 18.93% of the verification time for `pyModelChecking`, 52.7% for NuSMV, and 98.83% for Spot, in mean weighted by total execution time. In the case of Spot, the preparatory time includes the construction within `umaudemc` of the Büchi automaton for the formula and the C++ object of the Kripke structure that is directly used by the model checker, so this should explain the high percentage of preparation time.

- NuSMV counterexamples are smaller than Maude's in the 38.3% of the cases and bigger in the 7.5%, with 3.4 fewer states in mean. Spot counterexamples are also smaller than Maude's in the 44.19% of the cases and bigger in the 5%, with 3.08 fewer states in mean. Spot has functionality for trying to reduce counterexamples that has not been used.

- The builtin Maude model checker, LTSmin, and Spot use the automata-theoretic approach for LTL (see Section 2.3.2), for what they generate a Büchi automaton from the formula. Their sizes coincide in almost every case, except for twelve formulae where the Spot's automaton is smaller than Maude's and LTSmin's, and for another property in which Maude's is smaller than LTSmin's in one state.

We can conclude that using the builtin Maude model checker is the most appropriate option for LTL formulae, although smaller counterexamples are more likely found with other backends. For branching-time formulae, the performance of the backends is similar and there is no clear choice. In general, `pyModelChecking` should not be recommended for complex problems, since it cannot stand with the most complex in this test suite. However, its performance for small examples is comparable, despite being a Python implementation of textbook model-checking algorithms for education purposes.

Part IV

Conclusions

Chapter 11

Conclusions

In this thesis we have further developed the Maude strategy language to control rewriting, studied the model-checking problem for strategy-controlled systems, built model checkers for Maude specifications controlled by that language, and used all of these tools to specify and verify various systems from different fields. Its main contributions are the following:

- The development of the Maude strategy language by its implementation as an integral part of Maude [DEE⁺20], extending its algebra of modules with strategy modules that offer the same possibilities as their functional and system counterparts, including reflection, compositionality, and parameterization [RMP⁺19c]. Described in Chapter 3, this is a useful product of this work and a prerequisite for the rest of the thesis, but it is not its most original contribution, since the design of the language was already established, the current implementation was already started by Steven Eker, and there was a previous Maude-based working although incomplete implementation [EMM⁺07]. However, we have extended the language with some unplanned and interesting features like its reflection at the metalevel [RMP⁺20a] and the strategy-restricted variant of the `search` command. The first allows powerful metaprogramming with strategies and extending the language at the user choice (see Chapter 7), and the second permits exploring the evolution of controlled systems at the rewrite level as well as checking invariants and solving reachability problems.

- Still in Chapter 3, we have improved the theoretical foundations of the strategy language by specifying a complete denotational semantics that unlike previous presentations takes care of recursive strategies, nontermination, and the finite failure involved in strategy conditionals. We have also defined an equivalent small-step operational semantics that completely describes the rewriting sequences allowed by a strategy, and not only the results of the strategy computations. Strategies can be portrayed using this semantics in its most abstract form as subsets of the executions of the model according to the notion of extensional strategies [BCD⁺09], very useful for understanding their model-checking problem. The language properties of strategies thus described have been studied to elucidate the expressivity of the language. Although these semantics are presented for a specific language, they can be easily adapted to other strategy languages in the rewriting and abstract reduction context, which we have found very similar in Section 3.8.

- We have developed several examples of strategy-controlled specifications within Maude in Part III, aiming at showing that strategies can be efficiently used to specify models while respecting the separation of concerns principle enunciated by Kowalski as *Algorithm* = *Logic* + *Control* [Kow79] and by Lescanne as the *Rule* + *Control* approach [Les90]. The most evident example is the refined description of several procedures for equational completion on the Bachmair and Dershowitz's inference rules in Section 6.5, which redoes a previous work [VM11] controlling exactly the same set of rules by different strategy modules to specify the Knuth-Bendix, N-completion, S-completion, and ANS-completion procedures. Concurrency problems, games, protocols, and programming languages have also been specified like this.

- The contribution that gives name to this thesis is the study of the model-checking problem for strategy-controlled systems in Chapter 4. We have made explicit in abstract and simple terms for a very general notion of strategy a natural intuition that implicitly appears in the semantics of strategic logics [MMP+14] and other verification tools [DJL+15]: a linear-time temporal property is satisfied by a strategy-controlled model if it is satisfied by all the executions allowed by the strategy, and a branching-time property is satisfied if it is valid in the execution tree pruned by the strategy. Temporal properties can be effectively model checked using standard algorithms for the logic of choice by transforming the model to embody its restrictions. This can always be done on finite Kripke structures if the strategy denotation is closed and ω-regular. Moreover, ω-regular linear-time properties like LTL formulae can be decided for arbitrary ω-regular strategies in PSPACE and even for context-free strategies. Model checking Maude specifications controlled by the strategy language is well-defined under the principles above and using its small-step operational semantics to denote the executions allowed by an expression. This has been implemented as an extension [RMP+19a] of the Maude LTL model checker [EMS04], whose performances are comparable as seen in Chapter 10.

- Looking for completeness, branching-time logics have been addressed in Section 4.4, the associated satisfaction and synthesis problem is studied in Section 4.6, and a generalization of the strategy language where iterations behave like the Kleene star is presented in Section 4.5. Checking branching-time properties requires refining the model transformation used for linear-time verification. We have implemented adaptations to support logics like CTL, CTL*, and μ-calculus through external model checkers. Standard Maude specifications can also be checked against these logics, for which there were no active tools available. Moreover, the infrastructure developed to connect Maude with external model checkers is useful and promising for many other applications. Namely, the `maude` package described in Appendix A is a general purpose library that allows manipulating Maude entities from external programming languages like Python. Even though this is not properly a research contribution, it notably facilitates experimentation and integrating Maude into other tools. The `umaudemc` utility that brings uniform access to all supported model checkers, and the less standard prototype for checking properties against the Kleene-star semantics of the iteration have been programmed using this library. Regarding the associated satisfaction and synthesis problem, we have seen that it is PSPACE-complete for logics like LTL.

- Specification of probabilistic systems and quantitative model checking are also handled in Chapter 5, where `umaudemc` has been extended to support inserting probabilistic information into Maude specifications and having them verified with the external probabilistic model checker PRISM [KNP11]. Alternatively, an extension of the Maude strategy language with two additional probabilistic operators can also be used to specify probabilities on top of an existing Maude specification, which can then be executed in stochastic simulations. This is the most recent work incorporated into the thesis and further developments are still in progress. However, what it is explained here is well-founded and useful for many applications.

- Finally, the model checkers have been used to verify properties on multiple strategy-based Maude specifications in Part III. The most relevant example is the strategy-based environment for the simulation and model checking of membrane systems [RMP+20b]. This distributed computational model is a paradigmatic use case for strategies, since its rules are applied locally in a parallel and controlled manner, and the global evolution of the system takes place in phases. Extensions for many variants of membrane systems have been implemented without much effort, and model checking against all logics supported by our tools is available for free as a consequence of specifying them as a strategy-controlled system. Parallel rewrites are handled seamlessly as atomic steps of the model thanks to the opaque strategies feature of the model checker, and this approach can be used for other examples of parallel rewriting, which is sometimes difficult to represent by other means. Multistrategies [RMP+20a] is another interesting metaexample that allows specifying distributed strategies for the actors, players, or components of the system that are combined by a global strategy either concurrently, by turns, or at the user choice.

We should recognize that most of the examples and properties we have checked are not specially realistic or large enough. However, other researchers have used our model checker to evaluate properties on actual smart contracts specified in the BitML language for the Bitcoin blockchain [ABL+19]. We hope that the efficient availability of the strategy language and of the model-checking tools encourages more people to use them for their work.

11.1 Future work

Some theoretical and practical aspects of this thesis can be further extended in a natural way. We may ask which is the exact complexity of the satisfaction problem for branching-time logics and implement prototypes for synthesizing strategies given a rewrite system and a formula, find how to transform μ-calculus formulae to enforce the Kleene-star semantics of the iteration, make explicit how under this semantics LTL properties could be encoded in the strategy and model checked, support more verification logics and techniques, profile and improve the performance of the strategy language, the model checker, and the connection with external tools, etc. Another interesting continuation is applying these tools to more relevant, realistic, or complex examples of systems and properties. Involving other specification paradigms and verification problems, more original extensions can be envisaged.

Performance. The implementation of the Maude strategy language is currently being modified in order to improve its performance on the aspects we have identified in the evaluation of Chapter 10. By using the static information about the strategy definitions of a module, we can reduce the operations on the execution stack and better plan the detection of redundant computations and cycles, which is time and memory consuming. Moreover, this new implementation will solve the small divergence of the implementation regarding the formal semantics, where cyclic computations within a strategy condition should disable the execution of the negative branch. Currently, we can say that the new approach simplifies the code specific to the model checker, but the benchmarks in Chapter 10 should be repeated and compared to measure the actual effects of the modification when finished.

Narrowing. Narrowing generalizes term rewriting by allowing free variables in terms and replacing matching with unification. Strategies can also be used to control narrowing [Pad87] under the same abstract theoretical framework we have presented in Section 2.2, and the resulting strategy-controlled models can be checked against temporal properties in much the same way. However, an efficient implementation should be adapted to this specific setting, where techniques like folding can greatly reduce the state space by merging solutions that are logically subsumed by others. Strategy-controlled narrowing and its model checking can be useful to handle symbolic reachability and verification problems. Currently, (uncontrolled) narrowing-based Maude specifications can be model checked using the Maude logical model checker [BEM13], and the application of the strategy language for narrowing is being addressed by Luis Aguirre [AMP+21].

Probabilistic models. While probabilistic models have already been addressed in Chapter 5, this preliminary work can still be completed and extended. For instance, the two approaches proposed in this chapter can be related with each other, allowing the simulation of the models produced by the probability assignment methods of the `umaudemc` tool, and the probabilistic model checking of strategy-controlled systems using the quantified **`choice`** combinator. Integrating the simulation environment of the extended strategy with statistical model checkers like MultiVeSta [SV13a] would be straightforward and have interesting applications. More examples should also be written to evaluate the advantages of these approaches.

Strategies for verification. Strategies are used in this thesis as a specification resource, but they can also be used as a verification tool. Large state spaces can be reduced using strategies to eliminate redundancies, select alternative subsets of executions, or guide the model checker to a conjectured counterexample. Moreover, counterexamples obtained from strategy-controlled verification can be more readable since they are formed in the strategy mold, as we have seen in Section 6.3. Linear-time properties and universally quantified branching-time formulae that are refuted in a restricted model do not hold for sure in the original one. However, the validity of a formula can only be concluded from the submodel after proving that the restriction is without loss of generality. This is related to the partial order reduction technique [Pel93]. This proof may have to be specific for each problem, but finding conditions applicable to common situations is an interesting subject.

Appendices

Appendix A

The `maude` library

As mentioned in Section 4.4, the unified model-checking tool `umaudemc`, its builtin model checkers and its connections with external ones, and some other tools operate with Maude using the `maude` Python library. This library is an object-oriented interface to Maude that allows manipulating terms, modules, and other Maude entities as Python objects, whose methods expose the operations and information available in the Maude interpreter. Hence, any program that requires interacting with Maude can benefit from this library. Some other programming languages are supported like Java and Lua, thanks to the SWIG interface generator [SWI20] that provides the necessary glue code from our specification of the intended interface.

The typical work with the library starts by loading some Maude files with the `load` function, obtaining a `Module` object by means of the `getModule` function, and parsing objects with `parseTerm`. Most commands available in the interpreter are available as methods of the `Term` class.

```python
import maude
maude.init()
maude.load('15puzzle')
m = maude.getModule('15PUZZLE')
t = m.parseTerm('2 * 3')
t.reduce()
print(t.getSort(), ':', t)
```

Term objects can also be obtained from other reflection levels using `upTerm`/`downTerm`, and they can be built out of their arguments using the `makeTerm` function of the `Symbol` class. All module entities like sorts, operators, membership axioms, equations, rules, and strategies can be inspected using the `Module` object, and iterating over the arguments of a term is also possible with its `arguments` method. These features allow reusing and analyzing the results of previous operations easily without using the metalevel.

Strategy expressions can also be parsed and executed as in the `srewrite`, `dsrewrite`, and `search` command through this library.

```python
s = m.parseStrategy('left | right')
for result, nrew in m.parseTerm('puzzle1').srewrite(s, depth=True):
  print(result, 'in', nrew, 'rewrites')
```

Although the Maude LTL model checker and our extension for strategy-controlled systems can be executed by reducing the corresponding `modelCheck` operator using the library, they have a dedicated interface giving access to their results and extended information programmatically. They are invoked by the `modelCheck` methods of the `RewriteGraph` and `StrategyRewriteGraph` objects, which receive the formula as a term and return a structure with the model-checking result, including the indices of the counterexample states within the graph. These graphs can also be used to explore the models via the `getNextState` and `getStateTerm` methods. This is what the `umaudemc` tool uses to generate the models for some external and internal model checkers.

The rest of the library API is described in its documentation [Rub21a], which also includes some more unconventional and experimental features like the possibility of defining special operators in the target language. Custom special operators make the communication bidirectional, like a foreign function interface for Maude, since reducing or rewriting such terms will trigger the execution of Python or other language code, whose results will come back to Maude. We have used this feature to allow random access to a big map in a robot navigation controller written in Maude [MMR+21]. In general, the whole library is suitable for many recurrent use cases regarding the integration of Maude in external programs, and we hope it will be useful for other applications.

The underlying communication between the target language and Maude takes place at the binary level, with objects and functions of the target language wrapping C++ objects and functions of the Maude implementation. This means that importing the `maude` package in Python loads Maude dynamically as a shared library along with the Python interpreter, like a headless instance of the Maude interpreter that is maintained during the whole execution of Python. This kind of libraries are usually called *language bindings*, and the glue code required to connect the original implementation with the target language interpreter or compiler is often generated using tools like SWIG. Since Maude does not currently have an official and stable programming interface in the C++ language, this binding makes direct use of its internal classes and methods. An alternative, which we have used before and that is still used by some other tools [MSC+13, MT04], is communicating with Maude through the standard input and output, by writing commands to the interpreter and parsing their results. However, the proposed library is a more efficient and robust approach. Moreover, the same interface would be in principle available to every language supported by SWIG, although specific adaptations are required to better fit the target language style.

Since we started the development of this library, the Maude interpreter has gained the ability to call external programs, reading files, etc. An official and stable C library and a foreign function interface have been brought up too as long-term goals to advance in the connection of Maude to the outside world.

Appendix B

Proofs

This appendix gathers the proofs for all the propositions appearing in this thesis. They are referenced by their order number. Only proof sketches are given for some routine inductive proofs or case distinctions.

B.1 The Maude strategy language

The aim of the first part of this section is proving Theorem 3.1, i.e. that the denotational semantics of Section 3.4 is well defined.

Proposition 3.1 (page 64). *$(\mathcal{P}_\perp(M), \leq)$ is a chain-complete partially ordered set. Its minimum is $\{\perp\}$, and its maximal elements are M and the final sets. The union of classes is well defined by the union of its representatives, and for any chain $F \subseteq \mathcal{P}(\mathcal{P}_\perp(M))$, $\sup F = \bigcup_{A \in F} A$ if $\perp \in A$ for all $A \in F$, and $\sup F = Z$ if there is a $Z \in F$ such that $\perp \notin Z$. In this case, it is unique.*

Proof. We have to check that $\leq$ is a partial order and that every chain has a supremum in $\mathcal{P}_\perp(M)$. For the first task, let $A, B, C \in \mathcal{P}_\perp(M)$.

- *Reflexive*: we must prove $A \leq A$. If $\perp \in A$ then $A \leq A \iff A \setminus \{\perp\} \subseteq A$ which is true. Otherwise, $\perp \notin A$ so $A \leq A \iff A = A$.
- *Transitive*: suppose $A \leq B$ and $B \leq C$. If $\perp \notin A$ then $A = B$ and $\perp \notin B$. If $\perp \notin B$ then $B = C$ so $A \leq C$. Otherwise, $A \setminus \{\perp\} \subseteq B$ and $B \setminus \{\perp\} \subseteq C$, so $A \setminus \{\perp\} \subseteq C$, which means $A \leq C$.
- *Antisymmetric*: suppose $A \leq B$ and $B \leq A$. If $\perp \notin A$ or $\perp \notin B$ then $A = B$ by definition of $\leq$. Otherwise and also from the definition, $A \setminus \{\perp\} \subseteq B$ and $B \setminus \{\perp\} \subseteq A$. Both A and B contain $\perp$ so $A \subseteq B$ and $B \subseteq A$, and then $A = B$.

We can conclude $\leq$ is a partial order. Let F be a chain. We consider two cases:

- *For all $A \in F$, $\perp \in A$.* We prove $S = \bigcup_{A \in F} A \in \mathcal{P}_\perp(M)$ is the supremum.
  - It is an upper bound. Take $A \in F$ and see $A \leq S$. In this case $\perp \in A$, so this is equivalent to $A \setminus \{\perp\} \subseteq S$ and this is true, because $A \subseteq S$.
  - It is the least upper bound, because any other $B \in \mathcal{P}_\perp(M)$ such that $\forall A \in F \;\; A \leq B$ satisfies $S \leq B$ or equivalently $S \setminus \{\perp\} \subseteq B$ because $\perp \in S$. For any $x \in S \setminus \{\perp\}$, since S is a union, there is an $A \in F$ such that $x \in A$. Since $A \setminus \{\perp\} \subseteq B$, $x \in B$, and we conclude $S \leq B$.

- *There is a $Z \in F$ such that $\perp \notin Z$*, then Z must be the supremum of F. For this to make sense there must be only one set such that $\perp \notin Z$. Suppose there were another set $\perp \notin A \in F$. Then $A \leq Z$ or $Z \leq A$ because F is a chain, in any case $A = Z$. There is only one.
  - Z is an upper bound. Take $A \in F$, let's see $A \leq Z$. F is a chain so $A \leq Z$ or $Z \leq A$. If the first holds we have finished. Otherwise, $Z = A$ provided $\perp \notin Z$ and consequently $A \leq Z$ too.
  - Since $Z \in F$, it must be the supremum.

□

Remember that a function $f : E \to F$ between two ccpos $(E, \leq)$ and $(F, \leq)$ is said to be *monotone* if $f(x) \leq f(y)$ for all $x \leq y$, and *continuous* if $\sup f(C) = f(\sup C)$ for any chain C in E. The product ccpo $(E \times F, \sqsubseteq)$ is defined by $(x_1, y_1) \sqsubseteq (x_2, y_2)$ if $x_1 \leq x_2$ and $y_1 \leq y_2$. Functions defined on products are monotone or continuous iff all the functions $g(x) = f(x, y)$ for fixed $y \in F$ and $g(y) = f(x, y)$ for fixed $x \in E$ are so. Similarly, functions with range on a product are monotone and continuous iff the projections of the function on each component are so. Functions whose range is a ccpo $f : X \to E$ are also a ccpo with the order $f \sqsubseteq g$ if $f(x) \leq g(x)$ for all $x \in X$. This construction can be seen as a generalization of the product.

Lemma B.1. *The function* let $: \mathcal{P}_\perp(N) \times (N \to \mathcal{P}_\perp(M)) \to \mathcal{P}_\perp(M)$ *defined in Section 3.4 is monotone and continuous.*

Proof. Proving the monotonicity and continuity of this function is a tedious manipulation of the order definition and its consequences. A detailed proof is available for this particular definition in [RMP+21c] and similar constructs can be found regarding power domains. Here, we only informally write a proof sketch. Intuitively, let(U, f) is the union of $f(x)$ for all $x \in U$ taking nontermination into account. Adding elements to U makes the union grow too, since more sets $f(x)$ are included. If some of these sets are replaced by supersets the same effect is achieved. However, greater elements in $\mathcal{P}_\perp$ can also be obtained by removing $\perp$ from them. If $\perp$ is removed from U, the base elements in the union will not change, and at most $\perp$ may disappear from the result, making it greater. If $\perp$ disappears from some operands $f(x)$, at most $\perp$ will disappear from the union, and the result will still be greater. Continuity follows similarly, since the supremum of a chain is essentially the union of its elements, and unions can be freely swapped. In chains containing a final set, which are necessarily finite, the supremum is not a union but the greatest element of the chain. Mononicity relates the maximum of the original chain with the maximum of its image. □

Lemma B.2. *The following functions* $\mathrm{SFun}^n \to \mathrm{SFun}$ *or* $\mathrm{SFun}^n \to \mathcal{P}_\perp(\mathrm{VEnv})$ *are continuous for any rule label rl,* $l, r, P \in T_\Sigma(X)$ *and condition* C*:*

1. $\mathit{Comp}(f, g) = g \circ f$
2. $\mathit{Union}(f, g) = (\theta, t) \mapsto f(\theta, t) \cup g(\theta, t)$
3. $\mathit{Star}(f) = (\theta, t) \mapsto \bigcup_{n=0}^{\infty} f^n(\theta, t) \cup \{\perp : f^n(\theta, t) \neq \emptyset \text{ for all } n \in \mathbb{N}\}$
4. $\mathit{Cond}(f, g, h) = (\theta, t) \mapsto h(\theta, t) \text{ if } f(\theta, t) = \emptyset \text{ else } (g \circ f)(\theta, t)$
5. $\mathit{Check}(C, \theta, f_1, \ldots, f_n) = \mathrm{check}(C, \theta, \alpha_1 \cdots \alpha_n)$ *with* $[\![\alpha_k]\!]$ *replaced by* f_k*. The same holds for the derived operators Mcheck, Xmcheck and Amcheck.*
6. $\mathit{ApplyRule}(f_1, \ldots, f_n) = (\theta, t) \mapsto \bigcup_{(rl,l,r,C) \in R} \text{let } (\sigma, c) \leftarrow \mathit{Amcheck}(l, t, C,$
$$\mathrm{id}[x_1/\theta(t_1), \ldots, x_m/\theta(t_m)]; f_1 \cdots f_n, \theta_s) : \{c(\sigma(r))\}.$$

7. *SubRewrite*$(f_1, \dots, f_n) = (\theta, t) \mapsto \bigcup_{\sigma \in \mathrm{mcheck}(P,t,C,\theta)}$
$$\text{let } t_1 \leftarrow f_1(\sigma, \sigma(x_1)), \dots, t_n \leftarrow f_n(\sigma, \sigma(x_n)) : \{\sigma[x_1/t_1, \dots, x_n/t_n](P)\}.$$

Proof. It is enough to see whether they are continuous for each argument separately, as mentioned in the paragraph before Lemma B.1. Moreover, when the range is SFun, it is enough to study their evaluation on a fixed element in the domain. In many cases we will use that the composition of continuous functions is continuous, and the evaluation $\mathrm{eval}_x(f) = f(x)$ on a fixed $x \in X$ is continuous too.

1. Since $(g \circ f)(x) = \mathrm{let}(f(x), g)$, *Comp* is continuous iff $H_x(f, g) \coloneqq \mathrm{let}(f(x), g)$ is continuous for all x in the domain of SFun. This is the classical composition $H_x = \mathrm{let} \circ h_x$ where $h_x(f, g) \coloneqq (f(x), g)$, and h_x is continuous because it is constant in f and the identity in g in the second factor, and constant in g and the evaluation in f in the first one.

2. The same arguments invoked for proving that let is continuous are applicable in a much more simple case to this union.

3. The monotonicity and continuity of the first clause follow by inductively applying the previous statements. This is clear for f^n and infinite unions do not pose any particular problem. The only difference with finite unions is that $\bot$ may become implicitly included in the resulting set if it is infinite, but then it will never disappear when the terms of the union grow. The second clause is also monotone and continuous, since it can only take the values $\{\bot\}$ and $\emptyset$. When it takes the second, $f^n(\theta, t) = \emptyset$ for some $n \in \mathbb{N}$, so $\bot \notin f^n(\theta, t)$ for all $n \in \mathbb{N}$ and every g with $f \sqsubseteq g$ would yield the same value.

4. The function *Cond* is clearly continuous on g and h for fixed f, because it is either the identity on h or the continuous function *Comp*(f, g). These case definitions are also continuous on f, and the only problem is the transition between cases. Regarding monotonicity, consider $f \sqsubseteq f'$ and suppose $f'(\theta, t) = \emptyset$, then $f(\theta, t)$ is either $\emptyset$ or $\{\bot\}$ by definition of the order. It is not possible that $f(\theta, t)$ is empty and $f'(\theta, t)$ is not because sets without $\bot$ cannot grow. In the first case, $f(\theta, t) = \emptyset$, there is no case transition and both evaluations fall into the empty definition, which is monotone. In the second case $f(\theta, t) = \{\bot\}$, *Comp*(f, g) can only be $\{\bot\}$ by definition of the composition, so *Comp*(f', g) will trivially be greater in the order since $\{\bot\}$ is the minimum.

   Regarding continuity, the problem is again the chains that transit from one definition to the other. However, the images of all these chains are necessarily the chain $\{\{\bot\}, h(\theta, t)\}$ whose supremum $h(\theta, t)$ coincides with the evaluation of *Cond* in the supremum of its argument.

5. We check monotonicity and continuity by induction on the structure of C.

   - The base case is *Check*$(\mathtt{true}, \sigma, f_1, \dots, f_n, \theta) = \{\sigma\}$ which is a constant in f_k. Hence, it is monotone and continuous.
   - For *Check*$(l = r \wedge C, c, f_1, \dots, f_n, \theta)$. If $\sigma(l) = \sigma(r)$, its value is *Check*$(C, \theta, f_1, \dots, f_n, \theta_s)$, which is continuous by induction hypothesis; otherwise, its definition is $\emptyset$, a continuous constant.
   - For *Check*$(l \coloneqq r \wedge C, \sigma, f_1, \dots, f_n, \theta) = \bigcup_{\sigma' \in \mathrm{match}(\sigma(l), \sigma(r))}$ *Check*$(C, \sigma' \circ \theta, f_1, \dots, f_n, \theta)$. All these recursive calls are continuous by induction hypothesis, and so is their union.
   - For the rewriting condition, *Check*$(l \Rightarrow r \wedge C, \theta, f_1, \dots, f_n, \theta_s)$

   $$\text{let } t \leftarrow f_1(\theta, r) : \bigcup_{\sigma' \in \mathrm{xmatch}(\theta(r), t)} \mathit{Check}(C, \sigma' \circ \theta, f_2 \cdots f_n, \theta)$$

   The terms involved in the union are continuous by induction hypothesis, their union and the composition with let too.

The derived check operators are finite unions of the previous sets for all substitutions from a match, so they are continuous.

6. *Amcheck* is continuous by the previous item and the body of the let is a constant on f_k. Their combination by the let operator and the union is continuous too.

7. Despite the syntactic sugar, the definition is the union of several composed let functions, which are continuous as in the other cases.

□

Theorem 3.1 (page 68). *For any strategy expression α, $[\![\alpha]\!]_{(d_1,\ldots,d_m)}$ is monotone and continuous in $d_1, \ldots, d_m$, and so is the operator $F : \mathrm{SFun}^m \to \mathrm{SFun}^m$*

$$F(d_1, \ldots, d_m) \coloneqq \left([\![\delta_1]\!]_{(d_1,\ldots,d_m)}, \ldots, [\![\delta_m]\!]_{(d_1,\ldots,d_m)} \right)$$

Hence, F has a least fixed point $\mathsf{FIX}\, F \in \mathrm{SFun}^m$ which can be calculated as

$$\mathsf{FIX}\, F = \sup \{ F^n(\{\bot\}, \ldots, \{\bot\}) : n \in \mathbb{N} \}$$

Proof. All semantic functions are well defined because they are given an unambiguous functional definition and all cases are covered.

According to Kleene's Fixed Point Theorem [Win93; Thm 5.11] , if F is continuous in SFun^m then it has a least fixed point calculated as above. In fact, monotonicity is a sufficient condition for the existence of a least fixed point according to Knaster-Tarski Fixed Point Theorem [Win93; §5.5] , but not to find the characterization we have written.

F is monotone and continuous if each component $F_k = [\![\delta_k]\!]$ is. We will prove it by induction on the structure of δ_k:

- For `idle`, `fail`, and the `match` alternatives, the semantic function is a constant on d_k. Then it is continuous.

- $[\![\alpha\,;\beta]\!] = \mathit{Comp}([\![\alpha]\!], [\![\beta]\!])$. Its arguments $[\![\alpha]\!]$ and $[\![\beta]\!]$ are continuous by induction hypothesis, and *Comp* is continuous by the previous lemma, so functional composition preserves both.

- The same argument applies to both $[\![\alpha\,|\,\beta]\!] = \mathit{Union}([\![\alpha]\!], [\![\beta]\!])$, $[\![\alpha^*]\!] = \mathit{Star}([\![\alpha]\!])$ and $[\![\alpha \ ?\ \beta : \gamma]\!] = \mathit{Cond}([\![\alpha]\!], [\![\beta]\!], [\![\gamma]\!])$.

- Rule application is exactly the *ApplyRule* function of the previous lemma, and `matchrew` coincides with *SubRewrite*. The denotations of their substrategies are continuous by induction hypothesis, and the composition is continuous too.

- For $[\![sl(t_1, \ldots, t_n)]\!](\theta, t) = f_{sl}(\theta(t_1), \ldots, \theta(t_n), t)$, and

  $$f_{sl}(\overline{s}, t) = \bigcup_{(sl, \overline{p}_k, \delta_k, C_k) \in D} \bigcup_{\sigma \in \mathrm{mmatch}(\overline{p}_k, \overline{s}, C_k)} d_k(\sigma, t).$$

  This is continuous and monotone on d_k.

□

Equivalence of the two semantics for the strategy language

Now we start the proof of equivalence between the previous denotational semantics in Section 3.4 and the operational semantics in Section 3.5.

Definition B.1. *We define the function* dsem $: \mathcal{XS} \to \mathcal{P}_\perp(T_\Sigma(X))$ *as follows*

1. $\mathrm{dsem}(t \,@\, \varepsilon) = \{t\}$
2. $\mathrm{dsem}(t \,@\, \theta s) = \mathrm{dsem}(t \,@\, s)$
3. $\mathrm{dsem}(t \,@\, \alpha s) = \mathrm{let}\ t' \leftarrow [\![\alpha]\!](\mathrm{vctx}(s), t) : \mathrm{dsem}(t' \,@\, s)$
4. $\mathrm{dsem}(\mathrm{subterm}(x_1 : q_1, \ldots, x_n : q_n; t) \,@\, s) = \mathrm{let}\ _{i=1}^{n}\ t_i \leftarrow \mathrm{dsem}(q_i) : \mathrm{dsem}(t[x_i/t_i]_{i=1}^n \,@\, s)$
5. $\mathrm{dsem}(\mathrm{rewc}(p : q, \sigma, C, \overline{\alpha}, \theta_s, r, c; t) \,@\, s) =$
$$\mathrm{let}\ t' \leftarrow \mathrm{dsem}(q) : \textstyle\bigcup_{\sigma_m \in \mathrm{match}(p,t')} \mathrm{let}\ \sigma' \leftarrow \mathrm{check}(C, \sigma_m \circ \sigma; \overline{\alpha}, \theta_s) : \{c(\sigma'(r))\}$$

Proposition B.1. *For* $\alpha, \beta \in \mathrm{Strat}$ *and* $s \in \mathrm{Stack}$

1. $\mathrm{dsem}(t \,@\, \alpha \,;\, \beta\, s) = \mathrm{dsem}(t \,@\, \alpha\beta s)$
2. $\mathrm{dsem}(t \,@\, \alpha \,|\, \beta\, s) = \mathrm{dsem}(t \,@\, \alpha s) \cup \mathrm{dsem}(t \,@\, \beta s)$
3. $\mathrm{dsem}(\mathrm{rewc}(p : t \,@\, \alpha\, \theta_s, \sigma, C, \overline{\alpha}, \theta_s, r, c; t) \,@\, s) = \mathrm{let}\ \sigma' \leftarrow \mathrm{check}(t' \Rightarrow p \wedge C, \sigma; \alpha\overline{\alpha}, \theta_s) :$ $\{m(\sigma'(r))\}$ *for* t' *such that* $\sigma(t') = t$.

Proof. A routine calculation proves the first statement (we write θ for $\mathrm{vctx}(s)$)

$$\begin{aligned}
\mathrm{dsem}(t \,@\, \alpha \,;\, \beta\, s) &= \mathrm{let}\ t' \leftarrow [\![\alpha \,;\, \beta]\!](\theta, t) : \mathrm{dsem}(t' \,@\, s) \\
&= \mathrm{let}\ t' \leftarrow (\mathrm{let}\ t_m \leftarrow [\![\alpha]\!](\theta, t) : [\![\beta]\!](\theta, t_m)) : \mathrm{dsem}(t' \,@\, s) \\
&= \mathrm{let}\ t_m \leftarrow [\![\alpha]\!](\theta, t) : (\mathrm{let}\ t' \leftarrow [\![\beta]\!](\theta, t_m) : \mathrm{dsem}(t' \,@\, s)) \\
&= \mathrm{let}\ t_m \leftarrow [\![\alpha]\!](\theta, t) : \mathrm{dsem}(t_m \,@\, \beta\, s)) \\
&= \mathrm{dsem}(t \,@\, \alpha\beta\, s)
\end{aligned}$$

and the second

$$\begin{aligned}
\mathrm{dsem}(t \,@\, \alpha \,|\, \beta\, s) &= \mathrm{let}\ t' \leftarrow [\![\alpha \,|\, \beta]\!](\theta, t) : \mathrm{dsem}(t' \,@\, s) \\
&= \mathrm{let}\ t' \leftarrow [\![\alpha]\!](\theta, t) \cup [\![\beta]\!](\theta, t) : \mathrm{dsem}(t' \,@\, s) \\
&= \mathrm{let}\ t' \leftarrow [\![\alpha]\!](\theta, t) : \mathrm{dsem}(t' \,@\, s) \cup \mathrm{let}\ t' \leftarrow [\![\beta]\!](\theta, t) : \mathrm{dsem}(t' \,@\, s) \\
&= \mathrm{dsem}(t' \,@\, \alpha\, s) \cup \mathrm{dsem}(t' \,@\, \beta\, s)
\end{aligned}$$

For the third statement,

$$\begin{aligned}
&\mathrm{dsem}(\mathrm{rewc}(p : t \,@\, \alpha\theta_s, \sigma, C, \overline{\alpha}, \theta_s, r, c; t) \,@\, s) \\
&\quad = \mathrm{let}\ t_m \leftarrow [\![\alpha]\!](\theta_s, t) : \bigcup_{\sigma_m \in \mathrm{match}(p, t_m)} \mathrm{let}\ \sigma' \leftarrow \mathrm{check}(C, \sigma_m \circ \sigma; \overline{\alpha}, \theta_s) : \{c(\sigma'(r))\} \\
&\quad = \mathrm{let}\ \sigma' \leftarrow \mathrm{check}(t' \Rightarrow p \wedge C, \sigma; \alpha\overline{\alpha}, \theta_s) : \{c(\sigma'(r))\}
\end{aligned}$$

by the definition of check. □

Proposition B.2. *The execution tree for* $\to_{s,c}$ *is finitary.*

Proof. In other words, the number of direct successors of any state $q \in \mathcal{XS}$ for $\to_{s,c}$ is finite. Terms like $t\,@\,\theta s$ have $t\,@\,s$ as the only successor. Looking at the rules of the semantics in Section 3.5, it is easy to conclude that the successors of $t\,@\,\alpha s$ are finitely many (no more than two reductions except for `matchrew` and rule application, which have a successor for each possible match, anyhow a finite number).

Reasoning inductively, structured states have a finite number of successors too. In the case of rewc, they are the replacement of the substate by its successors (a finite number by induction hypothesis) or the rewc term for the next rewriting fragment or the plain term $t\,@\,s$ after successful evaluation of the condition. In the last two cases the successors are a finite amount because matches are finitely many. In the case of subterm, the states that may follow are the replaced term $t[x_i/t_i]\,@\,s$ upon successful termination, and the evolution of the subterms. In the last case the number of possibilities is the sum of the possibilities for all of its subterms, a finite number by induction hypothesis. □

On the contrary, the execution tree for $\twoheadrightarrow$ may not be finitary. Suppose two rules are defined for integer numbers: $n \Rightarrow s(n)$ labeled inc and $n \Rightarrow -m$ if $n \to m$ labeled neg. The execution state $0\,@\,neg\{inc^*\}$ has $-n\,@\,\varepsilon$ as successor for all $n \in \mathbb{N}$. In terms of $\to_{s,c}$ there is an infinite execution

$$0\,@\,neg\{inc^*\} \to_c \ldots \to_c \text{rewc}(m : k\,@\,inc^*, [n \mapsto 0], \texttt{true}, \varepsilon, \text{id}, -m; 0)\,@\,\varepsilon \to_c \ldots$$

Corollary B.1. *The execution tree for $\to_{s,c}$ is infinite iff it contains an infinite execution.*

Proof. The *if* case is obvious and the *only if* is Kőnig's lemma, since this tree is finitary by the previous lemma. □

Definition B.2. *The* call depth cdepth *of a state $q \in \mathcal{XS}$ is defined as:*

1. *For $t\,@\,s$, the number of substitutions in s.*
2. *For* subterm$(x_1 : q_1, \ldots, x_n : q_n; t)\,@\,s$*, the number of substitutions in s plus the maximum of zero and* cdepth$(q_i) - 1$ *for all i.*
3. *For* rewc$(p : q, \sigma, C, \overline{\alpha}, \theta_s, r, c; t)\,@\,s$*, the number of variable environments in s plus the maximum of zero and* cdepth$(q) - 1$.

The call depth *of an execution $q_1 \to_{s,c} q_2 \to^*_{s,c} q_n \to^*_{s,c} \cdots$ of length $1 \leq l \leq \infty$ is*

$$\sup\ \{\ \text{cdepth}(q_m) - \text{cdepth}(q_n) \mid 1 \leq n \leq l, n \leq m \leq l\ \}$$

or the maximum call depth of the executions in the proofs for the [else] rules if this is greater. It can be infinity.

The call depth is a measure of the number of nested calls, like the call stack size in a computer. One is subtracted to the call depths of the substates in items 2 and 3 above because the substitutions at their bottoms never come from a strategy call but from the matching environment of a `matchrew`, or the replication of the outer context in rewc. The call depth of a state increases when more context is added to the bottom of its queue, but adding context to all elements of an execution does not affect its call depth, since it is a difference. Another property is that the call depth of the suffixes of any execution are always less or equal than the depth of the whole.

For convenience and abusing notation, we write $q\,@\,s$ to represent a state in any of its forms with a global stack s. That is, $q\,@\,s$ can be $t\,@\,s$, subterm$(\ldots)\,@\,s$ or rewc$(\ldots)\,@\,s$.

Lemma B.3. 1. *If $t\,@\,s_1\,\theta \to^*_{s,c} t_m\,@\,\varepsilon$ and $t_m\,@\,s_2 \to^*_{s,c} q$ then $t\,@\,s_1 s_2 \to^*_{s,c} q$ where $\theta =$* vctx(s_2).

Moreover, the length of the resulting execution is the sum of the lengths of the original ones, and its call depth is the maximum.

2. *If $q_0 \,@\, s_1 s_2 \to^*_{s,c} q'$ and $s_1 \neq \varepsilon$ then*

$$q' \in \{q \,@\, s' s_2 \,:\, q_0 \,@\, s_1\theta \to^*_{s,c} q \,@\, s'\} \cup \{q \,:\, \exists\, t_m \quad q_0 \,@\, s_1\theta \to^*_{s,c} t_m \,@\, \varepsilon \,\wedge\, t_m \,@\, s_2 \to^*_{s,c} q\}$$

3. *If $t \,@\, s_1 s_2 \to^*_{s,c} t' \,@\, \varepsilon$ then there is a term t_m such that $t \,@\, s_1\theta \to^*_{s,c} t_m \,@\, \varepsilon$ and $t_m \,@\, s_2 \to^*_{s,c} t' \,@\, \varepsilon$.*

Proof. Some common facts follow easily from the inspection of the rules and axioms of the semantics definition:

- The global stack s in $q \,@\, s$, only changes by a $\to_{s,c}$ reduction if $q = t \,@\, s$ for some term t.
- Only the top of the stack can be popped by a reduction, $t \,@\, \alpha\, s \to_{s,c} q \,@\, s_\alpha\, s$ for some stack s_α.
- Reductions only depend on the top of the stack and the variable environment. Thus, the stack can be extended from below without effect if the latter is preserved, i.e. $q \,@\, s\,\theta \to_{s,c} q' \,@\, s'$ implies $q \,@\, s\, s_0 \to_{s,c} q \,@\, s'\, s_0$ where $\theta = \mathrm{vctx}(s_0)$, which may be omitted if it is id.

Using these basic facts, we prove the statements:

1. Replace θ by s_2 in the stacks' bottoms of the first execution. Then we obtain $t \,@\, s_1 s_2 \to^*_{s,c} t_m \,@\, s_2$ and joining it with the second execution, the statement holds.

   While the property about the length is obvious, we will discuss the call depth of the combined execution. It is the following maximum (the sup is a max because both executions are finite)
   $$\max\ \{\ \mathrm{cdepth}(q_m) - \mathrm{cdepth}(q_n) \mid 1 \leq n \leq l, n \leq m \leq l\ \}$$
   Let l_1 be the length of the first execution. For $n > l_1$, i.e. for states to the right of $t_m \,@\, s_2$, the numbers $\mathrm{cdepth}(q_m) - \mathrm{cdepth}(q_n)$ are the same as in the original execution because the states are exactly the same. The states before $t_m \,@\, s_2$ have been extended with s_2 at the bottom of their stacks, so their call depth is greater than that of $t_m \,@\, s_2$. Hence, terms like $\mathrm{cdepth}(q_m) - \mathrm{cdepth}(q_n)$ with $n < l_1$ and $m \geq l_1$ can be ignored when taking the maximum because they are bounded above by $\mathrm{cdepth}(q_m) - \mathrm{cdepth}(q_{l_1})$. The only missing case is $\mathrm{cdepth}(q_m) - \mathrm{cdepth}(q_n)$ with both $n, m < l_1$ but they are at most (and reach) the call depth of the first execution because they are the states of this execution extended uniformly at the bottom.

2. The proof will be carried out by induction on the length k of the derivation $q \,@\, s_1 s_2 \to^*_{s,c} q'$

   - Base case ($k = 0$): then $q_0 \,@\, s_1 s_2 = q'$, so q' is in the first set with the 0-length execution and $s' = s_1$.
   - Inductive case: we have $q_0 \,@\, s_1 s_2 \to_{s,c} q_1 \to^k_{s,c} q'$. If q_0 is a subterm or rewc state, the stack remains unchanged in the first step, so we can apply induction hypothesis on $q_1 \,@\, s_1 s_2 \to^k_{s,c} q'$ to conclude. Otherwise, $q_0 \,@\, s_1 s_2 = t \,@\, s_1 s_2$ for some term t. And since the stack s_1 is not empty, it should be either $\sigma s_1'$ or $\alpha\, s_1'$.
     In the first case $t \,@\, \sigma s_1' s_2 \to_c t \,@\, s_1' s_2$: if $s_1' = \varepsilon$ then q' is in the second set with $t_m = t$. Otherwise, we apply induction hypothesis to the rest of the derivation $t \,@\, s_1' s_2 \to^k_{s,c} q'$. Here, we have two possibilities:
     - There is a term t_m such that $t \,@\, s_1' s_2 \to^*_{s,c} t_m \,@\, s_2$ and $t_m \,@\, s_2 \to^*_{s,c} q'$. Hence, our q' is in the second set with $t_m = t$. This follows from the second execution above and from $t \,@\, \sigma s_1 \theta \to_c t \,@\, s_1\theta \to_c t \,@\, s_1'\theta \to^*_{s,c} t_m \,@\, \theta \to_c t_m \,@\, \varepsilon$.
     - $t \,@\, s_1'\theta \to^*_{s,c} q \,@\, s'$ for some state q and $q' = q \,@\, s' s_2$. Thus, q' is in the first set with $t \,@\, s_1\theta = t \,@\, \sigma s_1'\, \theta \to_c t \,@\, s_1'\, \theta \to^*_{s,c} q \,@\, s'$.

     The second case $t \,@\, \alpha s_1' s_2 \to_{s,c} q_1 \,@\, s_\alpha s_1' s_2$ is almost identical.

3. We only have to apply (2) with $q_0 @ s_1 s_2 = t @ s_1 s_2$ and $q' = t' @ \varepsilon$ if $s_1 \neq \varepsilon$ (otherwise is trivial).

□

Lemma B.4. *Given $f : \mathrm{VEnv} \times T_\Sigma(X) \to \mathcal{P}_\perp(T_\Sigma(X))$, assume*

$$\forall t, t' \in T_\Sigma(X) \quad \forall \theta \in \mathrm{VEnv} \qquad t' \in f(\theta, t) \implies t @ \alpha\, \theta \to^*_{s,c} t' @ \varepsilon$$

For any $k \in \mathbb{N}$, if $t' \in f^k(\theta, t)$ then $t @ \alpha^*\, \theta \to^*_{s,c} t' @ \varepsilon$ with k applications of the iteration rule.

Proof. By induction on k. If $k = 0$ (base case), $f^0(\theta, t) = \{t\}$ so $t = t'$ and we are done using the rule $t @ \alpha^*\, \theta \to_c t @ \theta \to_c t @ \varepsilon$. In the inductive case, assume the lemma holds for k. We want to prove that if $t' \in f^{k+1}(\theta, t)$ then $t @ \alpha^*\, \theta \to^*_{s,c} t' @ \varepsilon$. The definition of composition and $f^{k+1} = f \circ f^k$ imply that there is a term t_m such that $t_m \in f^k(\theta, t)$ and $t' \in f(\theta, t_m)$. Using, in this order, the second rule for the iteration, the assumption of the statement, and the induction hypothesis, we conclude

$$t @ \alpha^*\, \theta \to_c t @ \alpha\alpha^*\, \theta \to^*_{s,c} t_m @ \alpha^*\, \theta \to^*_{s,c} t' @ \varepsilon$$

□

Lemma B.5. *Given $f : \mathrm{VEnv} \times T_\Sigma(X) \to \mathcal{P}_\perp(T_\Sigma(X))$, assume*

$$\forall t, t' \in T_\Sigma(X) \quad \forall \theta \in \mathrm{VEnv} \qquad t @ \alpha\, \theta \to^*_{s,c} t' @ \varepsilon \implies t' \in f(\theta, t)$$

If $t @ \alpha^*\, \theta \to^*_{s,c} t' @ \varepsilon$ and the number of states in the execution with stack $\alpha^*\, \theta$ is exactly $k + 1$, then $t' \in f^k(\theta, t)$.

Proof. By induction on k. If $k = 0$, the only allowed execution is $t @ \alpha^*\, \theta \to_c t @ \theta \to_c t @ \theta$. So $t = t'$ and $t \in f^0(\theta, t) = \{t\}$. If $k > 0$, we can decompose the execution in two parts: before the penultimate state with stack $\alpha^*\, \theta$ and after this state. The first part is $t @ \alpha^*\, \theta \to^*_{s,c} t_m @ \alpha^*\, \theta$ and it can be completed with $t_m @ \alpha^*\, \theta \to_c t_m @ \theta \to_c t_m @ \varepsilon$. It contains exactly k states with $\alpha^*\, \theta$, so by induction hypothesis, $t_m \in f^{k-1}(\theta, t)$. The second execution is $t_m @ \alpha^*\, \theta \to^*_{s,c} t' @ \varepsilon$, but its second state must be $t_m @ \alpha\alpha^*\, \theta$ and the last states need be $t' @ \alpha^*\, \theta$ and $t' @ \theta$. Forgetting about the first and last two states and removing α^* from the stacks, we get $t_m @ \alpha\, \theta \to^*_{s,c} t' @ \varepsilon$. By the hypothesis of the lemma, $t \in f(\theta, t_m)$. Combining both results, $t' \in f(\theta, t_m)$ and $t_m \in f^{k-1}(\theta, t)$, we conclude $t' \in f^k(\theta, t)$ as we wanted. □

Lemma B.6. *Given $f : \mathrm{VEnv} \times T_\Sigma(X) \to \mathcal{P}_\perp(T_\Sigma(X))$ and assuming $\perp \in f^k(\theta, t)$ for some $k \in \mathbb{N}$, there is a term $t_\perp$ and $k_\perp \leq k$ such that $t_\perp \in f^{k_\perp}(\theta, t)$ and $\perp \in f(\theta, t_\perp)$.*

Proof. The proof is by induction on k. The premises never hold for $k = 0$, since $f^0(\theta, t) = \{t\}$. For $k = 1$, the statement clearly holds with $t_\perp = t$ and $k_\perp = 0$. For $k \geq 2$, remember $f^k(\theta, t) = \text{let } t_m \leftarrow f^{k-1}(\theta, t) : f(\theta, t_m)$. The following situations can happen:

- $\perp \in f(\theta, t_m)$. Then $t_\perp = t_m$ and $k_\perp = k - 1$.
- $\perp \in f^{k-1}(\theta, t)$. Then the induction hypothesis proves the lemma.

□

Lemma B.7. *Let $\mathrm{len}(q)$ denote the maximum possible length of an execution starting at q. Then*

1. $\mathrm{subterm}(x_1 : q_1, \ldots, x_n : q_n; t) @ \varepsilon \to^*_{s,c} \mathrm{subterm}(x_1 : q'_1, \ldots, x_n : q'_n; t) @ \varepsilon$ *if $q_i \to^*_{s,c} q'_i$ for all i.*

2. *If* $\mathrm{subterm}(x_1 : q_1, \dots, x_n : q_n; t) \mathbin{@} \varepsilon \to^*_{s,c} \mathrm{subterm}(x_1 : q'_1, \dots, x_n : q'_n; t) \mathbin{@} \varepsilon$ *then* $q_i \to^*_{s,c} q'_i$ *for all* i.
3. *If* $q_i \to^*_{s,c} t_i \mathbin{@} \varepsilon$ *for all* $1 \leq i \leq n$ *then* $\mathrm{subterm}(x_1 : q_1, \dots, x_n : q_n; t) \mathbin{@} \varepsilon \to^*_{s,c} t[x_i/t_i]_{i=1}^n \mathbin{@} \varepsilon$.
4. *If* $\mathrm{subterm}(x_1 : q_1, \dots, x_n : q_n; t) \mathbin{@} \varepsilon \to^*_{s,c} t' \mathbin{@} \varepsilon$ *then there are terms* $t_1, \dots, t_n$ *such that* $q_i \to^*_{s,c} t_i \mathbin{@} \varepsilon$ *for all* $1 \leq i \leq n$ *and* $t' = t[x_i/t_i]_{i=1}^n$.
5. $\mathrm{len}(\mathrm{subterm}(x_1 : q_1, \dots, x_n : q_n; t) \mathbin{@} \varepsilon) = \sum_{k=1}^n \mathrm{len}(q_k) + 1$.

Proof. Statements (3) and (4) are a corollary of statements (1) and (2). Induction is used to prove the latter.

1. By induction on the execution lengths from q_i, suppose all lengths are 0, then $q_i = q'_i$ and the empty execution solves the statement. Otherwise, there is at least a positive number in the n-tuple of execution lengths. We assume the statement is true for any collection of derivations of lower or equal length with at least one strictly lower coordinate. Suppose, without loss of generality, that $q_1 \to_{s,c} q_{1,1} \to^*_{s,c} q'_1$. Then by the rules that allow executing steps on the substates of subterm we can assert

$$\mathrm{subterm}(x_1 : q_1, \dots, x_n : q_n; t) \mathbin{@} \varepsilon \to_{s,c} \mathrm{subterm}(x_1 : q_{1,1}, \dots, x_n : q_n; t) \mathbin{@} \varepsilon$$

   And the remaining execution to be applied in x_1 is strictly shorter than the original one, while the rest have not changed. The derivation can be completed by induction hypothesis.

2. Now, induction will be based on the length of execution in the premise. Suppose the length is 0, then $q_i = q'_i$ and the empty executions from every q_i make the statement true. If the length is positive, there must be an initial transition inside a subterm

$$\begin{aligned} &\mathrm{subterm}(x_1 : q_1, \dots, x_i : q_i, \dots, x_n : q_n; t) \mathbin{@} \varepsilon \\ \to_{s,c}\ &\mathrm{subterm}(x_1, q_1, \dots, x_i : q_{i,1}, \dots, x_n : q_n; t) \mathbin{@} \varepsilon \end{aligned}$$

   and for this to be true $q_i \to_{s,c} q_{i,1}$ must hold. By induction hypothesis on the execution from the righthand side, $q_j \to^*_{s,c} q'_j$ for all $j \neq i$ and $q_i \to_{s,c} q_{i,1} \to^*_{s,c} q'_i$. So we have found the desired executions.

3. It is enough to apply (1), then $\mathrm{subterm}(x_1 : t_1 \mathbin{@} \varepsilon, \dots, x_n : t_n \mathbin{@} \varepsilon; t) \mathbin{@} \varepsilon \to_c t[x_i/t_i]_{i=1}^n \mathbin{@} \varepsilon$.
4. $t[x_i/t_i]_{i=1}^n \mathbin{@} \varepsilon$ must be preceded by $\mathrm{subterm}(x_1 : t_1 \mathbin{@} \varepsilon, \dots, x_n : t_n \mathbin{@} \varepsilon; t) \mathbin{@} \varepsilon$ so that we can apply (2) to conclude.
5. For the last statement, we can observe in the proofs of the previous items that the size of the subterm execution we have built in (1,3) is the sum of the lengths of their subexecutions plus the ending transition. And the length of the substate executions we have recovered in (2,4) sum the length of the original derivation except for the last $\to_c$ transition. Then we conclude respectively the $\geq$ and the $\leq$ of the fifth statement, so we can assert the identity.

□

Lemma B.8. *For any conditions* C_1 *and* C_2, *and* $\overline{\alpha}_k \in \mathrm{Strat}^*$ *for the rewriting fragments of* C_k,

$$\mathrm{check}(C_1 \wedge C_2, \sigma, \overline{\alpha}_1\overline{\alpha}_2, \theta) = \mathrm{let}\ \sigma' \leftarrow \mathrm{check}(C_1, \sigma, \overline{\alpha}_1, \theta) : \mathrm{check}(C_2, \sigma', \overline{\alpha}_2, \theta)$$

and if C_1 *is equational* ($\overline{\alpha}_1 = \varepsilon$), *the* let *is a normal union.*

Proof. Follows from the recursive definition of check. □

Generalizing the semantic equivalence statement of Theorem 3.2, we should prove that $t \in \operatorname{dsem}(q)$ is safisfied iff $q \to_{s,c}^{*} t \mathbin{@} \varepsilon$, and that $\operatorname{dsem}(q)$ contains $\perp$ iff there is an infinite execution by $\to_{s,c}$ from q. The simultaneous proof of both $\Rightarrow$ and $\Leftarrow$ will be done by induction, on the structure of q but also on the approximants of the semantic function. Remember that the meaning of the strategy definitions in a module is given by functions d_i, and d_i is defined as the i-th component of the fixed point $\mathsf{FIX}\, F$ of the function $F(d_1, \ldots, d_{|D|}) = (\llbracket \delta_1 \rrbracket, \ldots, \llbracket \delta_{|D|} \rrbracket)$ where δ_i is the strategy definition term. By Theorem 3.1 we know that $\mathsf{FIX}\, F = \sup\{F^n(\{\perp\}, \ldots, \{\perp\}) : n \in \mathbb{N}\}$, where the supremum is the union (perhaps removing $\perp$) by coordinates. We write $d_k^{(n)} = F_k^n(\{\perp\}, \ldots, \{\perp\})$. And replacing d_k by $d_k^{(n)}$, we also define $\llbracket \alpha \rrbracket^{(n)}$, $\operatorname{dsem}^{(n)}$ and $\operatorname{check}^{(n)}$. Observe that

$$d_k^{(n+1)} = F_k(d_1^{(n)}, \ldots, d_{|D|}^{(n)}) = \llbracket \delta_k \rrbracket (d_1^{(n)}, \ldots, d_{|D|}^{(n)})$$

so $d_k^{(n)} \sqsubseteq d_k^{(n+1)}$ because F is monotonic. Its derived functions are monotone too. The following technical lemma will be the main part of the proof.

Lemma B.9. *The property* $\mathrm{p}(n, q)$ *defined by the following statements is satisfied for all* $n \in \mathbb{N}$ *and execution state* q*:*

1. *Forall* $1 \leq k \leq |D|$, $t \in d_k^{(n)}(\theta, t_0)$ *implies* $t_0 \mathbin{@} \delta_k\, \theta \to_{s,c}^{*} t \mathbin{@} \varepsilon$.
2. *Forall* $t \in \operatorname{dsem}^{(n)}(q)$, $q \to_{s,c}^{*} t \mathbin{@} \varepsilon$.
3. *If* $q \to_{s,c}^{*} t \mathbin{@} \varepsilon$ *with call depth at most* n*, then* $t \in \operatorname{dsem}^{(n)}(q)$.
4. *If* $\perp \in \operatorname{dsem}^{(n)}(q)$*, then there is a* $\to_{s,c}$ *execution containing infinitely many consecutive iterations of the same strategy or whose call depth is at least* $n+1$ *ignoring [else] proofs.*
5. *If* $\perp \notin \operatorname{dsem}^{(n)}(q)$*, then all executions from* q *are finite and its call depth is at most* n.

Proof. The induction property $\mathrm{p}(n, q)$ in $\mathbb{N} \times \mathcal{XS}$ will be proved by induction on the lexicographic order for the well-defined $\mathbb{N}^4$ tuple

$$(n, \text{strategy constructors in } q, \text{nested } \mathcal{XS} \text{ constructors in } q, \text{condition fragments in } q)$$

It is clearly a well-founded order, and every syntactic substate of an execution state is below the whole state.

The first statement is different from the rest because it does not start with an execution state q but a semantic denotation $d_k^{(n)}$. Hence, we prove it first for any $q \in \mathcal{XS}$ using a separate inductive argument.

- For $n = 0$, $d_k^{(0)} = \{\perp\}$ (the constant function) so $t \in d_k^{(0)}(\theta, t_0)$ never holds and (1) is true vacuously.
- Now assume $\mathrm{p}(n, q)$ for all $q \in \mathcal{XS}$ to prove the first statement of $\mathrm{p}(n+1, q)$. If $t \in d_k^{(n+1)}(\theta, t_0)$, and by the recursive definition of the approximants, $t \in \llbracket \delta_i \rrbracket^{(n)}(\theta, t_0) = \operatorname{dsem}^{(n)}(t_0 \mathbin{@} \delta_i\, \theta)$. The induction hypothesis (item 2) let us conclude $t_0 \mathbin{@} \delta_i\, \theta \to_{s,c}^{*} t \mathbin{@} \varepsilon$.

Now, the rest of the statements are proven ranging over q for any $n \in \mathbb{N}$, starting from the stack cases. We will usually make explicit which elements are in the semantics $\operatorname{dsem}^{(n)}$ and which derivation can follow from the state, so that the properties follow easily. When dealing with (4), the considerations about infinite iterations will not be made explicit except in the iteration case, because this property is trivially inherited by the composed executions. The base cases are:

- `idle`. From one side $\operatorname{dsem}^{(n)}(t_0 \mathbin{@} \texttt{idle}\, \theta) = \llbracket \texttt{idle} \rrbracket^{(n)}(\theta, t_0) = \llbracket \texttt{idle} \rrbracket(\theta, t_0) = \{t_0\}$, and from the other side, $t_0 \mathbin{@} \texttt{idle}\, \theta \to_c t_0 \mathbin{@} \theta \to_c t_0 \mathbin{@} \varepsilon$ is the only allowed execution from $t_0 \mathbin{@} \texttt{idle}\, \theta$. It is clear that (2), (3) and (5) hold. Statement (4) holds trivially.

- `fail`. We know $\mathrm{dsem}^{(n)}(t_0 \mathbin{@} \mathtt{fail}\, \theta) = [\![\mathtt{fail}]\!]^{(n)}(\theta, t_0) = [\![\mathtt{fail}]\!](\theta, t_0) = \varnothing$ and that no $\to_{s,c}$ transition starts from $t_0 \mathbin{@} \mathtt{fail}\, \theta$. So all statements are satisfied vacuously.

- `match`. Again

$$\mathrm{dsem}^{(n)}(t_0 \mathbin{@} \mathtt{match}\, P\, \mathtt{s.t.}\, C\, \theta) = [\![\mathtt{match}\, P\, \mathtt{s.t.}\, C]\!]^{(n)}(\theta, t_0) = [\![\mathtt{match}\, P\, \mathtt{s.t.}\, C]\!](\theta, t_0)$$

  which equals $\varnothing$ if $\mathrm{mcheck}(P, t, \theta) \neq \varnothing$ or $\{t_0\}$ otherwise. From the other side,

$$t_0 \mathbin{@} \mathtt{match}\, P\, \mathtt{s.t.}\, C\, \theta \to_c t_0 \mathbin{@} \theta \to_c t_0 \mathbin{@} \varepsilon$$

  if the matching is not empty, and no transition leaves the state otherwise. All this implies (2), (3) and (5), while (4) is trivially true.

- $rl\,[x_1 \leftarrow t_1, \ldots, x_m \leftarrow t_m]$. Now

$$\begin{aligned}
\mathrm{dsem}^{(n)}(t_0 \mathbin{@} rl\,[x_1 \leftarrow t_1, \ldots, x_m \leftarrow t_m]\, \theta) &= [\![rl\,[x_1 \leftarrow t_1, \ldots, x_m \leftarrow t_m]]\!]^{(n)}(\theta, t_0) \\
&= [\![rl\,[x_1 \leftarrow t_1, \ldots, x_m \leftarrow t_m]]\!](\theta, t_0) \\
&= \mathrm{ruleApply}(t_0, rl, \mathrm{id}[x_i/\theta(t_i)]_{i=1}^n)
\end{aligned}$$

  and iff t belongs to such a set we have

$$t_0 \mathbin{@} rl\,[x_1 \leftarrow t_1, \ldots, x_m \leftarrow t_m]\, \theta \to_s t \mathbin{@} \theta \to_c t \mathbin{@} \varepsilon$$

  and this implies (2), (3) and (5). The premise of (4) cannot happen.

And now the inductive cases. We will sometimes resort to Proposition B.1 and understand that it is extended to the superscripted dsem because the proof is exactly the same.

- $rl\,[x_1 \leftarrow t_1, \ldots, x_m \leftarrow t_m]\{\alpha_1, \ldots, \alpha_k\}$. In this case, the rule for controlled rule applications is the only applicable.

$$\begin{gathered}
t \mathbin{@} rl\,[x_1 \leftarrow t_1, \ldots, x_m \leftarrow t_m]\{\alpha_1, \ldots, \alpha_k\}\, \theta \\
\to_c \mathrm{rewc}(r_1 : \sigma(l_1) \mathbin{@} \alpha_1 \theta, \sigma, C, \alpha_2 \cdots \alpha_k, r, m; t) \mathbin{@} \theta
\end{gathered}$$

  for any rule $(rl, l, r, C_0 \wedge l_1 \Rightarrow r_1 \wedge C)$ and match $(\sigma, c) \in \mathrm{amatch}(l, t, \mathrm{id}[x_i/\theta(t_i)]_{i=1}^m, C_0)$. The second state is lower in the order than the first (a strategy constructor less), so induction hypothesis can be applied on it. By Proposition B.1, the $\mathrm{dsem}^{(n)}$ of both sides is the same. Properties (2), (3), (4), and (5) follow immediately, by only prepending the initial state.

- $\theta\, s$. The only allowed reduction here is $t_0 \mathbin{@} \theta s \to_c t_0 \mathbin{@} s$ and, by definition of dsem, the semantics of both sides are exactly the same. Properties (2) and (5) follow directly.

  (3). Given a derivation $t_0 \mathbin{@} \theta s \to_c t_0 \mathbin{@} s \to_{s,c}^* t \mathbin{@} \varepsilon$ whose call depth is at most n, the suffix derivation from $t_0 \mathbin{@} s$ has lower or equal call depth, just like any other sub-derivation. So the induction hypothesis $\mathrm{p}(n, t_0 \mathbin{@} s)$ can be applied to conclude that $t \in \mathrm{dsem}^{(n)}(t_0 \mathbin{@} \theta\, s)$.

  (4). If $\perp$ is in the $\mathrm{dsem}^{(n)}$, there is an execution from $t_0 \mathbin{@} s$ with call depth at least $n+1$, so the full execution has call depth at least $n+1$ because it must be greater or equal than that of its suffix.

- $\alpha\, s$, where s contains at least a strategy expression. We know

$$R \coloneqq \mathrm{dsem}^{(n)}(t_0 \mathbin{@} \alpha\, s) = \mathrm{let}\ t_m \leftarrow [\![\alpha]\!]^{(n)}(\theta, t_0) : \mathrm{dsem}(t_m \mathbin{@} s)$$

  and we can invoke the induction hypotheses $\mathrm{p}(n, t_0 \mathbin{@} \alpha\, \theta)$ and $\mathrm{p}(n, t_m \mathbin{@} s)$ where $\theta = \mathrm{vctx}(s)$ (both are lower in the order because the number of strategy constructors decreases).

(2). From the induction hypotheses, $t_0 \,@\, \alpha\, \theta \to^*_{s,c} t_m \,@\, \varepsilon$ and $t_m \,@\, s \to^*_{s,c} t \,@\, \varepsilon$ if $t \in \text{dsem}(t_0 \,@\, \alpha\, s)$. These executions can be joined by Lemma B.3 (item 1) to build the desired one.

(3). We are give an execution $t_0 \,@\, \alpha\, s \to^*_{s,c} t \,@\, \varepsilon$. This execution can be split using the third statement of Lemma B.3 in two executions $t_0 \,@\, \alpha\, \theta \to^*_{s,c} t_m \,@\, \varepsilon$ and $t_m \,@\, s \to^*_{s,c} t \,@\, \varepsilon$ for some term t_m (both clearly have lower or equal call depth than the original one). The induction hypotheses say that $t_m \in \text{dsem}^{(n)}(t_0 \,@\, \alpha\, \theta) = [\![\alpha]\!]^{(n)}(\theta, t_m)$ and $t \in \text{dsem}(t_m \,@\, s)$, so we can conclude $t \in \text{dsem}(t_0 \,@\, \alpha s)$.

(4). $\perp \in R$ if $\perp \in [\![\alpha]\!]^{(n)}(\theta, t_0)$ or $\perp \in \text{dsem}(t_m \,@\, s)$ for some t_m. By induction hypothesis and in any case, the combined execution's call depth is at least $n + 1$ because one of its components is.

(5). If $\perp \notin R$, $\perp$ is neither in $\text{dsem}^{(n)}(t_m \,@\, s)$ for any t_m in $[\![\alpha]\!]^{(n)}(\theta, t_0)$ nor in $[\![\alpha]\!]^{(n)}(\theta, t_0)$ itself, so all executions from these states are finite and their call depth is at most n by Lemma B.3 (1).

- $\alpha\, ;\beta$. By Proposition B.1, $t_0 \,@\, \alpha\, ;\beta\, \theta$ and $t_0 \,@\, \alpha\beta\, \theta$ are semantically equivalent, $t_0 \,@\, \alpha\, ;\beta\, \theta \to_c t_0 \,@\, \alpha\beta\, \theta$ is the single possible reduction from the state, and $t_0 \,@\, \alpha\beta\, \theta$ is below $t_0 \,@\, \alpha\, ;\beta\, \theta$ in the order (the sequence constructor is removed). So this case is reduced to the previous one, using the induction hypothesis.

- $\alpha \mid \beta$. The possible successors are

$$\begin{tikzcd}[row sep=tiny] & t_0 \,@\, \alpha\, \theta \\ t_0 \,@\, \alpha \mid \beta\, \theta \arrow[ur, "c"] \arrow[dr, "c"'] & \\ & t_0 \,@\, \beta\, \theta \end{tikzcd}$$

and by Proposition B.1, $\text{dsem}^{(n)}(t_0 \,@\, \alpha \mid \beta\, \theta) = \text{dsem}^{(n)}(t_0 \,@\, \alpha\, \theta) \cup \text{dsem}^{(n)}(t_0 \,@\, \beta\, \theta)$.

(2). Suppose without loss of generality that $t \in \text{dsem}^{(n)}(t_0 \,@\, \alpha\, \theta)$. By $\text{p}(n, t_0 \,@\, \alpha\, \theta)$, $t_0 \,@\, \alpha\, \theta \to^*_{s,c} t \,@\, \varepsilon$. Prefixing this execution with the initial state by the disjunction rule, we obtain an execution from $t_0 \,@\, \alpha \mid \beta\, s$ to $t \,@\, \varepsilon$ as we wanted.

(3). Any $t_0 \,@\, \alpha \mid \beta\, \theta \to^*_{s,c} t \,@\, \varepsilon$ must start with a $\to_c$ transition to $t_0 \,@\, \alpha\, \theta$ or $t_0 \,@\, \beta\, \theta$. Without loss of generality, suppose we are in the second case. By induction hypothesis $\text{p}(n, t_0 \,@\, \beta\, \theta)$, $t \in \text{dsem}^{(n)}(t_0 \,@\, \beta\, \theta)$ but this is included in the $\text{dsem}^{(n)}$ set of the initial term.

(4). If $\perp$ is in the $\text{dsem}^{(n)}$ of the initial set, it must be in the semantic set of one of its descendants, so there is an execution with call depth at least n from it. Prepending $t_0 \,@\, \alpha \mid \beta\, \theta$ as initial state does not affect the call depth, so we have (4).

(5). All derivations from the successor terms above being finite, the derivation from the disjunction only increases the length in one.

- α^*. The possible successors are

$$\begin{tikzcd}[row sep=tiny] & t_0 \,@\, \theta \arrow[r, "c"] & t_0 \,@\, \varepsilon \\ t_0 \,@\, \alpha^*\, \theta \arrow[ur, "c"] \arrow[dr, "c"'] & & \\ & t_0 \,@\, \alpha\alpha^*\, \theta & \end{tikzcd}$$

and if $t \in \text{dsem}^{(n)}(t_0 \,@\, \alpha^* \theta) = [\![\alpha^*]\!]^{(n)}(\theta, t_0)$ then there is a $k \in \mathbb{N}$ with $t \in {[\![\alpha]\!]^{(n)}}^k(\theta, t_0)$. We can always use the induction hypothesis $\text{p}(n, t_0 \,@\, \alpha\, \theta)$.

(2). By the induction hypothesis and Lemma B.4 we obtain (2).

(3). Take an execution $t_0 \mathbin{@} \alpha^* \, \theta \to_{s,c}^* t' \mathbin{@} \varepsilon$. An inductive proof will be sketched. There are only two possibilities for the second state: $t_0 \mathbin{@} \theta$, in which case the only reachable t is t_0 and it belongs to $[\![\alpha^*]\!]^{(n)}(\theta, t_0)$, and $t_0 \mathbin{@} \alpha\alpha^* \, \theta$. In this case, and using Lemma B.3 (item 3), there must be a term t_m such that $t_0 \mathbin{@} \alpha \, \theta \to_{s,c}^* t_m \mathbin{@} \varepsilon$ and $t_m \mathbin{@} \alpha^* \theta \to_{s,c}^* t \mathbin{@} \varepsilon$. The call depths of both executions are at most n, so the induction hypothesis $\mathrm{p}(n, t_0 \mathbin{@} \alpha \, \theta)$ lets us conclude that $t_m \in [\![\alpha]\!]^{(n)}(\theta, t_0)$ and, since $t_m \mathbin{@} \alpha^* \theta \to_{s,c}^* t' \mathbin{@} \varepsilon$ is strictly shorter than the original execution, we can assert $t \in [\![\alpha^*]\!]^{(n)}(\theta, t_m)$. By these two memberships and the semantic definition of α^*, $t \in [\![\alpha^*]\!]^{(n)}(\theta, t_0)$.

(4). If $\bot \in \mathrm{dsem}^{(n)}(t_0 \mathbin{@} \alpha^* \, \theta) = [\![\alpha^*]\!]^{(n)}(\theta, t)$, $\bot$ is the denotation of some iteration of the body or infinitely many iterations are allowed for α. In the first case, by Lemma B.6 there is a $k_\bot \in \mathbb{N}$ and a term $t_\bot \in [\![\alpha]\!]^{(n),k_\bot}(\theta, t)$ such that $\bot \in [\![\alpha]\!]^{(n)}(\theta, t_\bot)$. By induction hypothesis and Lemma B.4, $t_0 \mathbin{@} \alpha^* \, \theta \to_{s,c}^* t_\bot \mathbin{@} \varepsilon$ and this must end with $t_\bot \mathbin{@} \alpha^* \, \theta \to_c t_\bot \mathbin{@} \theta \to_c t_\bot \mathbin{@} \varepsilon$. Using induction hypothesis $\mathrm{p}(n, t_\bot \mathbin{@} \alpha \, \theta)$, there is an execution whose call depth is at least $n+1$ from $t_\bot \mathbin{@} \alpha \, \theta$. Assembling both using Lemma B.3 (item 1) and other rules and extensions, we conclude with

$$t_0 \mathbin{@} \alpha^* \, \theta \to_{s,c}^* t_\bot \mathbin{@} \alpha^* \, \theta \to_c t_\bot \mathbin{@} \alpha\alpha^* \, \theta \to_{s,c}^* \cdots$$

whose call depth is at least $n+1$.

If not in the case of the paragraph above and $[\![\alpha]\!]^{(n)^k}(\theta, t) \neq \varnothing$ for all $j \in \mathbb{N}$, there are executions with k applications of the iteration rule for all $k \in \mathbb{N}$ by Lemma B.4. According to statement (5) of $\mathrm{p}(n, t \mathbin{@} \alpha \, \theta)$ for each t, the terms that can be reached after an iteration are finitely many because the possible executions are finite. The tree whose root is t_0 and whose children are the terms that can be reached after a single iteration is then finitary and infinite, and by Kőnig's lemma it contains an infinite branch, and so there is an infinite sequence of iterations (an infinite execution would be easier to prove thanks to Corollary B.1).

(5). Then, if $\bot \notin \mathrm{dsem}^{(n)}(t_0 \mathbin{@} \alpha^* \, \theta)$ we are sure there is a k_0 such that $[\![\alpha]\!]^{(n),k_0}(\theta, t_0) = \varnothing$. Moreover, for any term t in the finite set $[\![\alpha^*]\!]^{(n)}(\theta, t_0)$, since $\bot \notin [\![\alpha]\!]^{(n)}(\theta, t)$, all reachable executions of the body are finite and their lengths are bounded. The first phrase implies there are no execution sequences that unroll α more than n_0 times, and so the maximum length of any execution is at most n_0 times the maximum length of an iteration, a finite number. The proof of the fact that α is not unrolled more that n_0 times is given by Lemma B.5 (its assumption is true by induction hypothesis $\mathrm{p}(n, t_0 \mathbin{@} \alpha \, \theta)$), because to unroll a $n_0 + 1$ time there must be a state $t \mathbin{@} \alpha^* \theta$ with $t \in [\![\alpha]\!]^{(n),k_0}(\theta, t) = \varnothing$ and this is impossible.

- $\alpha ? \beta : \gamma$. The possible successors are these two, but the second line can only be taken if the execution tree from $t_0 \mathbin{@} \alpha \, \theta$ is finite and does not contain solutions

$$\begin{tikzcd} & t_0 \mathbin{@} \alpha\beta \, \theta \\ t_0 \mathbin{@} \alpha ? \beta : \gamma \, \theta \arrow[ur, "c"] \arrow[r, "c"'] & t_0 \mathbin{@} \gamma \, \theta \end{tikzcd}$$

And $\mathrm{dsem}^{(n)}(t_0 \mathbin{@} \alpha ? \beta : \gamma \, \theta) = [\![\alpha ? \beta : \gamma]\!]^{(n)}(\theta, t_0)$ is $[\![\beta]\!]^{(n)} \circ [\![\alpha]\!]^{(n)}(\theta, t_0)$ if $[\![\alpha]\!]^{(n)}(\theta, t_0) \neq \varnothing$ and $[\![\gamma]\!]^{(n)}(\theta, t_0)$ otherwise.

(2). In the first case, since $t \in [\![\beta]\!]^{(n)} \circ [\![\alpha]\!]^{(n)}(\theta, t_0)$, there is a term t_m such that $t_m \in [\![\alpha]\!]^{(n)}(\theta, t_0)$ and $t \in [\![\beta]\!]^{(n)}(\theta, t_m)$. By the induction hypotheses $\mathrm{p}(n, t_0 \mathbin{@} \alpha \, \theta)$ and $\mathrm{p}(n, t_m \mathbin{@} \beta \, \theta)$, there are executions $t_0 \mathbin{@} \alpha \, \theta \to_{s,c}^* t_m \mathbin{@} \varepsilon$ and $t_m \mathbin{@} \beta \, \theta \to_{s,c}^* t \mathbin{@} \varepsilon$. By Lemma B.3, $t_0 \mathbin{@} \alpha\beta \, \theta \to_{s,c}^* t_m \mathbin{@} \beta \, \theta \to_{s,c}^* t \mathbin{@} \varepsilon$ as we wanted.

In the second case, by $\mathrm{p}(n, t_0 \mathbin{@} \alpha\, \theta)$ and $\mathrm{p}(n, t_0 \mathbin{@} \gamma\, \theta)$, $t_0 \mathbin{@} \gamma\, \theta \to^*_{s,c} t \mathbin{@} \varepsilon$, to which we append the [else] rule application because the execution tree from α is finite without solutions. Indeed, $\mathrm{dsem}^{(n)}(t_0 \mathbin{@} \alpha\, \theta) = [\![\alpha]\!]^{(n)}(\theta, t_0) = \varnothing$ and by property (5) every execution from $t_0 \mathbin{@} \alpha\, \theta$ is finite with call depth at most n, and by property (3) and since no $t \in \varnothing$, no execution from α leads to a solution.

(3). Take a derivation $t_0 \mathbin{@} \alpha ? \beta : \gamma\, \theta \to^*_{s,c} t \mathbin{@} \varepsilon$. The second state can be $t_0 \mathbin{@} \alpha\beta\, \theta$ or $t_0 \mathbin{@} \gamma\, \theta$, in the latter case with a finite execution tree without solutions from $t \mathbin{@} \alpha\, \theta$. Anyhow we can apply induction hypothesis to the sequent (less strategy constructors). In the first case t is in $\mathrm{dsem}^{(n)}(t_0 \mathbin{@} \alpha\beta\, \theta)$ and this set is contained in the $\mathrm{dsem}^{(n)}$ value of the original state, so t is in it.
In the second case, we know by (3) in $\mathrm{p}(n, t_0 \mathbin{@} \gamma\, \theta)$ that $t \in [\![\gamma]\!]^{(n)}(\theta, t_0)$. We want to prove that $R := [\![\alpha]\!]^{(n)}(\theta, t_0)$ is empty, to apply the semantic definition for the conditional and finally conclude that t belongs to its semantic set. If $\bot \notin R$, by property (5) of $\mathrm{p}(n, t_0 \mathbin{@} \alpha\, \theta)$ every derivation from α has call depth at most n, and by property (3), no term belongs to R. Without terms and without $\bot$, $R = \varnothing$. Otherwise $\bot \in R$, and there is an execution from α whose call depth is at least $n + 1$. Hence, the execution tree makes the initial execution call depth greater than n, against the hypothesis. This latter case is not possible.

(4). If $\bot$ is in the global $\mathrm{dsem}^{(n)}$, $\bot \in [\![\alpha]\!]^{(n)}$, or $[\![\alpha]\!]^{(n)} \neq \varnothing$ and $\bot \in [\![\beta]\!]^{(n)}$, or $[\![\alpha]\!]^{(n)} = \varnothing$ and $\bot \in [\![\gamma]\!]^{(n)}$ with the appropriate arguments. In any case and by the induction hypothesis we have invoked before, an execution whose call depth is greater than n can be built, in some cases using Lemma B.3.

(5). If $\bot$ is not a member of the global dsem, all executions from α are finite and their call depth is at most n, and if $[\![\alpha]\!]^{(n)} \neq \varnothing$ then all executions from β are finite, and if $[\![\alpha]\!]^{(n)} = \varnothing$ then all executions from γ are finite. The executions from $t_0 \mathbin{@} \alpha ? \beta : \gamma\, \theta$ are only prepended by this initial state, so they are also finite. The same arguments are valid for the call depth.

- `matchrew` P `s.t` C `by` x_1 `using` $\alpha_1, \ldots, x_n$ `using` α_n. First, all possible reductions are

$$
\begin{aligned}
&t_0 \mathbin{@} \texttt{matchrew}\ P\ \texttt{s.t}\ C\ \texttt{by}\ x_1\ \texttt{using}\ \alpha_1, \ldots, x_n\ \texttt{using}\ \alpha_n\, \theta \\
&\quad \to_c \mathrm{subterm}(x_1 : \sigma(x_1) \mathbin{@} \alpha_1\sigma, \ldots, x_n : \sigma(x_n) \mathbin{@} \alpha_n\sigma; \sigma_{-\{x_1,\ldots,x_n\}}(P)) \mathbin{@} \theta
\end{aligned}
$$

for any $\sigma \in \mathrm{mcheck}(P, t_0, C, \theta)$. The righthand state is smaller than the lefthand state (a strategy constructor less), so we can apply the induction hypothesis. From the other side, the semantics of the first state equals the semantics of the second by the definition of the dsem, since the semantics of the substates $\mathrm{dsem}^{(n)}(q_i) = \mathrm{dsem}^{(n)}(\sigma(x_i) \mathbin{@} \alpha_i\sigma) = [\![\alpha_i]\!]^{(n)}(\sigma, \sigma(x_i))$ and both let expressions coincide. Their call depths are also the same. Thus, (2), (3), (4), and (5) immediately follow.

- $sl(t_1, \ldots, t_n)$. The only possible successors of $t_0 \mathbin{@} sl(t_1, \ldots, t_n)\, \theta$ are $t_0 \mathbin{@} \delta_i\, \sigma\, \theta$ for any δ_i and $\sigma \in \mathrm{mmatch}(\overline{p}_i, \overline{s}, C)$ where $\overline{s} = \theta(t_1) \cdots \theta(t_n)$. From the other side

$$
\begin{aligned}
\mathrm{dsem}^{(n)}(t_0 \mathbin{@} sl(t_1, \ldots, t_n)\, \theta) &= [\![sl(t_1, \ldots, t_n)]\!]^{(n)}(\theta, t_0) \\
&= \bigcup_{(sl, \overline{p}_i, \delta_i, C_i) \in D} \bigcup \left\{ d_i^{(n)}(\sigma, t_0) : \sigma \in \mathrm{mmatch}(\overline{p}_i, \overline{s}, C_i) \right\}
\end{aligned}
$$

Hence, $t \in \mathrm{dsem}^{(n)}(t_0 \mathbin{@} sl(t_1, \ldots, t_m)\, \theta)$ if $t \in d_i^{(n)}(\sigma, t_0)$ for some $1 \leq i \leq |D|$.

(2). By the property (1) we have inductively (but separately) proved in the first paragraphs and since $t \in d_i^{(n)}(\sigma, t_0)$, there must be an execution $t_0 \mathbin{@} \delta_i\, \sigma \to^*_{s,c} t \mathbin{@} \varepsilon$. θ can be added in the stacks' bottom, and using the call rule, we prove (2)

$$
t_0 \mathbin{@} sl(t_1, \ldots, t_n)\, \theta \to_c t_0 \mathbin{@} \delta_i\, \sigma\, \theta \to^*_{s,c} t \mathbin{@} \theta \to_c t \mathbin{@} \varepsilon
$$

The rest of the properties are not proven uniformly in n and we will distinguish cases. First suppose $n = 0$. Then $d_k^{(0)} = \{\bot\}$. So $\mathrm{dsem}^{(0)}(t_0 \mathbin{@} sl(t_1, \dots, t_n)\,\theta) = \{\bot\}$ except in case $m_{sl} = 0$ or no definition matches the call arguments, when it is $\emptyset$.

(3). It holds vacuously because every execution from $t_0 \mathbin{@} sl\,(t_1, \dots, t_n)\,\theta$ has call depth at least 1 and only executions whose call depth is 0 are considered.

(4). For the premise to be satisfied, we must be in the case where the denotation is $\{\bot\}$. The reduction $t_0 \mathbin{@} sl\,(t_1, \dots, t_n)\,\theta \to_c t_0 \mathbin{@} \delta_i\,\sigma\,\theta$ is allowed and its call depth is $1 > 0$.

(5). The statement holds vacuously except in the case where no definition is applicable and the denotation is $\emptyset$. The only possible execution is the initial state alone, which is finite and its call depth is zero.

Now, we prove the case $n > 0$. By definition $d_i^{(n+1)} = F_i(d_1^{(n)}, \dots, d_{|D|}^{(n)}) = [\![\delta_i]\!](d_1^{(n)}, \dots, d_{|D|}^{(n)})$, and also by definition

$$d_i^{(n+1)}(\sigma, t_0) = [\![\delta_i]\!]^{(n)}(\sigma, t_0) = \mathrm{dsem}^{(n)}(t_0 \mathbin{@} \delta_i \sigma)$$

We invoke $\mathrm{p}(n, t_0 \mathbin{@} \delta_i\,\sigma)$ and proceed

(3). Take an execution $t_0 \mathbin{@} sl\,(t_1, \dots, t_n)\,\theta \to_{s,c}^* t \mathbin{@} \varepsilon$ whose call depth is at most $n + 1$. The second state must be $t_0 \mathbin{@} \delta_i\,\sigma\,\theta$ for some i and θ, and its tail must be $t \mathbin{@} \sigma\,\theta \to_c t \mathbin{@} \theta \to_c t \mathbin{@} \varepsilon$. The maximum $\mathrm{cdepth}(q_m) - \mathrm{cdepth}(q_n)$ (with $m \geq n$) in the definition of call depth is reached only with $n = 1$ because the call depth is 1 in the initial state, while it is at least two in the rest of the execution but the two last states, where it only decreases. Thus, the execution $t_0 \mathbin{@} \delta_i\,\sigma \to_{s,c}^* t \mathbin{@} \varepsilon$, removing the last states and θ from the bottom, has a call depth of n. Hence, the induction hypothesis lets us conclude $t \in \mathrm{dsem}^{(n)}(t_0 \mathbin{@} sl\,(t_1, \dots, t_n)\,\theta)$.

(4). If $\bot \in [\![sl\,(t_1, \dots, t_n)]\!]^{(n+1)}(\theta, t_0)$ then by the definition recalled above $\bot \in d_i^{(n+1)}(\sigma, t_0)$ for some i and σ. So $\bot \in \mathrm{dsem}^{(n)}(t_0 \mathbin{@} \delta_i\,\sigma)$ and there is an execution $t_0 \mathbin{@} \delta_i\,\sigma \to_{s,c}^* q \mathbin{@} s$ with call depth at least $n + 1$ by induction hypothesis on n. Suppose the call depth is reached as the difference between that of state number u and that of the state number l with $u \geq l$. The call depth of all states is greater or equal than 1 due to σ except if $s = \varepsilon$, in which case the penultimate state is $q \mathbin{@} \sigma$. Hence, the call depth for l is at least 1, because if l were the last state, $u = l$, and the call depth would be 0 that cannot be $n + 1$ for any $n \in \mathbb{N}$. In conclusion, the call depth of q_u is at least $n + 2$. Let's build an execution with call depth at least $n+2$. Adding θ to the bottoms, the call depth of all states is increased by 1, and then we prepend the initial state to obtain

$$t_0 \mathbin{@} sl\,(t_1, \dots, t_n)\,\theta \to_c t_0 \mathbin{@} \delta_i\,\sigma\,\theta \to_{s,c}^* q \mathbin{@} s\,\theta.$$

Clearly the maximum in the call depth definition is reached subtracting the call depth of the first state, because 1 is the least we can substract and the maximum for $\mathrm{cdepth}(q_m)$ can be chosen in the greatest possible set. The call depth of the modified state u (now in position $u + 1$) has increased in one, so, subtracting 1, the call depth of the new execution is at least $n + 2$.

(5). If $\bot \notin [\![sl\,(t_1, \dots, t_n)]\!]^{(n+1)}(\theta, t_0)$, then $\bot \notin \mathrm{dsem}^{(n)}(t_0 \mathbin{@} \delta_i\,\sigma)$ for all i and match σ, so from these states all executions are finite (if any). The execution sequences starting from $t_0 \mathbin{@} sl\,(t_1, \dots, t_n)\,\theta$ are the previous ones prepended by $t_0 \mathbin{@} sl\,(t_1, \dots, t_n)\,\theta \to_c$, and adding θ at the bottom of their stacks. They can also be extended by $t \mathbin{@} \theta \to_c t \mathbin{@} \varepsilon$ to drop the new context at the end. Anyhow, they are still finite. This process also increases their call depth in 1, so that they are at most $n + 1$, which is valid. This can be justified as in (4).

- subterm$(\ldots, x_i : q_i, \ldots; t_0) @ \varepsilon$. Let q be this state in the following. We know

$$\text{dsem}^{(n)}(q) = \text{let } t_1 \leftarrow \text{dsem}^{(n)}(q_1) \ldots \text{let } t_k \leftarrow \text{dsem}^{(n)}(q_k) : \text{dsem}^{(n)}(t_0[x_i/t_i]_{i=1}^k @ \varepsilon)$$

So if $t \in \text{dsem}^{(n)}(q)$ there must be $t_i \in \text{dsem}^{(n)}(q_i)$ for all $1 \leq i \leq k$ such that $t = t_0[x_i/t_i]_{i=1}^k$.

(2). By induction hypotheses $\text{p}(n, q_i)$ for every substate q_i of q, $q_i \to_{s,c}^* t_i @ \varepsilon$. Lemma B.7 lets us conclude (2) because there is an execution $q \to_{s,c}^* t_0[x_i/t_i]_{i=1}^k @ \varepsilon = t @ \varepsilon$.

(3). Suppose there is an execution $q \to_{s,c}^* t @ \varepsilon$. By Lemma B.7 we know $q_i \to_{s,c}^* t_i @ \varepsilon$ for some terms t_i and $t = t_0[x_i/t_i]_{i=1}^k$. Moreover, the call depths of these nested executions are below n if the call depth of the whole execution is below n. Then by the induction hypotheses $\text{p}(n, q_i)$, $t_i \in \text{dsem}^{(n)}(q_i)$. Looking at the $\text{dsem}^{(n)}$ definition above and provided $\text{dsem}^{(n)}(t @ \varepsilon) = \{t\}$ we conclude $t \in \text{dsem}^{(n)}(q)$.

(4). If $\perp \in \text{dsem}^{(n)}(q)$ then $\perp \in \text{dsem}^{(n)}(q_l)$ for some $1 \leq l \leq k$. Then by hypothesis there is an execution $q_l \to_{s,c}^* q'$ whose call depth is at least $n + 1$. We can then build a derivation from q advancing the l-th component by the reductions of the aforementioned execution using the rules for executing steps inside subterms (Lemma B.7, item 1). Then, a derivation from q whose call depth is at least $n + 1$ exists.

(5). If $\perp \notin \text{dsem}^{(n)}(q)$, $\perp$ cannot be in any of the $\text{dsem}^{(n)}(q_i)$ by definition of let. Hence, all derivations from q_i are finite and so are those from q according to Lemma B.7. The condition about the call depths is also preserved.

- rewc$(p : q; \ldots; t_0)$. The semantics of this state is

$$\begin{aligned} R &:= \text{dsem}^{(n)}(\text{rewc}(p : q; \ldots; t_0) @ \theta) \\ &= \text{let } t' \leftarrow \text{dsem}^{(n)}(q) : \bigcup_{\sigma_m \in \text{match}(p,t')} \text{let } \sigma' \leftarrow \text{check}^{(n)}(C, \sigma_m; \overline{\alpha}, \theta) : \{m(\sigma'(r))\} \end{aligned}$$

If t belongs to the set above, there must be a $t_m \in \text{dsem}^{(n)}(q)$, σ_m, and σ'. From the other side, the execution from the term can evolve using the rules that apply steps inside the substate

$$\text{rewc}(p : q, \ldots) \to_{s,c}^* \text{rewc}(p : q', \ldots) \iff q \to_{s,c}^* q'$$

and if q' is a solution $t_m @ \varepsilon$, the following state must be one of two:

- C is equational and there is a match σ'. The following state is $t @ \theta \to_c t @ \varepsilon$ for some t.
- If C contains a rewriting fragment, $\text{rewc}(rc : \sigma_1(l) @ \alpha\, \theta, \sigma_1, C' \ldots) @ \theta$ is the next state, where $C = C_0 \wedge l \Rightarrow r \wedge C'$ with C_0 an equational condition and σ_1 like in the last control rule. This state is below in the order because it has fewer condition fragments, so we can call $\text{p}(n, \text{rewc}(rc : \sigma_1(l) @ \alpha\theta, \sigma_1, C', \ldots))$.
  More precisely, $\sigma_1 \in \text{check}(C_0, \sigma \circ \sigma_m)$, and according to Proposition B.1 (3) the semantics of this new execution state is let $\sigma' \leftarrow \text{check}^{(n)}(l \Rightarrow r \wedge C', \sigma_1; \overline{\alpha}, \theta)$: $\{m(\sigma'(r))\}$. By Lemma B.8, $\sigma' \in \text{check}^{(n)}(C, \sigma_m \circ \sigma; \overline{\alpha}, \theta)$. Hence, all reachable states from the new rewriting state are in the semantics of the initial execution state.

(2). Since $t \in \text{dsem}^{(n)}(q)$ and by induction hypothesis $\text{p}(n, q)$, $q \to_{s,c}^* t_m @ \varepsilon$ so

$$\text{rewc}(p : q, \sigma, C, \overline{\alpha}, \ldots) @ \theta \to_{s,c}^* \text{rewc}(p : t_m @ \varepsilon, \sigma, C, \overline{\alpha}, \ldots) @ \theta$$

and there are $\sigma_m \in \text{match}(p, t_m)$ and $\sigma' \in \text{check}^{(n)}(C, \sigma_m \circ \sigma; \overline{\alpha}, \theta)$ such that $t = m(\sigma'(r))$. Using the previous and if C is equational, we can apply the rule we have mentioned above and expand this derivation with $\to_s t\,@\,\theta \to_c t\,@\,\varepsilon$ as we wanted. Otherwise, if $C = C_0 \wedge l \Rightarrow rc \wedge C'$ and $\sigma' \in \mathrm{check}^{(n)}(C, \sigma_m \circ \sigma; \overline{\alpha}, \theta)$ there must be a $\sigma_1 \in \mathrm{check}(C_0, \sigma_m \circ \sigma)$ by Lemma B.8 such that $\sigma' \in \mathrm{check}^{(n)}(C', \sigma_1, \overline{\alpha}, \theta)$. So we can apply the rule cited above and append at the end of the execution $\mathrm{rewc}(rc : \sigma_1(l)\,@\,\alpha\theta, \sigma_1, C', \ldots)$. Applying the induction hypothesis to this state, we get the rest of the derivation and conclude.

(3). Take a derivation $\mathrm{rewc}(p : q, \ldots) \to_c t\,@\,\varepsilon$. There must be a first state $\mathrm{rewc}(p : t_m\,@\,\varepsilon, \ldots)$ and $q \to^*_{s,c} t_m\,@\,\varepsilon$. Hence and by induction hypothesis, $t_m \in \mathrm{dsem}^{(n)}(q)$. If the next state is a term state $t\,@\,\theta$, the system rule must have been applied so there is a σ' to conclude that $t \in \mathrm{dsem}^{(n)}(\mathrm{rewc}(\ldots))$.

When the next state is another rewriting condition state, the control rule must have been applied. So there are σ_m and σ_1. According to what we have said in the initial paragraphs of this case, all terms in $\mathrm{dsem}^{(n)}$ of the new state are in $\mathrm{dsem}^{(n)}$ of the initial state. And by induction hypothesis, the reachable solutions from the state are in its $\mathrm{dsem}^{(n)}$.

(4). If $\perp \in R$ then $\perp \in \mathrm{dsem}^{(n)}(q)$ or $\perp \in \mathrm{check}^{(n)}(C, \sigma_m \circ \sigma; \overline{\alpha}, \theta)$ (in which case there must be a rewriting condition fragment in C). If the first is true, there is an execution from q whose call depth is at least $n + 1$. Using the compositonal rule for rewc, we can easily build an execution from the rewc state with the same call depth.

In the second case $\perp$ is in the $\mathrm{dsem}^{(n)}$ of the next condition fragment state, so by induction hypothesis there is an execution from it whose call depth is at least $n + 1$, and prefixing the initial state preserves the call depth. So we have an execution from the initial state with a call depth greater than n.

(5). If $\perp \notin R$, all executions from q are finite with call depth at most n. If C does not contain rewriting fragments, the executions' length can only be extended by two new states whose call depths decrease. Otherwise, all derivations get stuck or arrive to the next rewriting fragment state. By induction hypothesis, all executions from it are finite and with call depth bounded by n, so the full execution also satisfies these properties.

□

Proposition B.3. *Forall $t \in T_\Sigma(X)$ and $q \in \mathcal{XS}$*

$$t \in \mathrm{dsem}(q) \quad\Longleftrightarrow\quad q \to^*_{s,c} t\,@\,\varepsilon$$

and $\perp \in \mathrm{dsem}(q)$ iff there is an infinite execution from q.

Proof. Each implication in the propisition follows directly from one of the statements of the property $\mathrm{p}(n, q)$ of Lemma B.9. Given an execution state q and a term t,

2. Given q, if $t \in \mathrm{dsem}(q)$, by the fixed-point definition recalled above, $t \in \mathrm{dsem}^{(n_0)}(q)$ for some $n_0 \in \mathbb{N}$. From $\mathrm{p}(n_0, q)$ we conclude $q \to^*_{s,c} t\,@\,\varepsilon$ ($\Rightarrow$) by item (2).

3. If we are given a derivation $q \to^*_{s,c} t\,@\,\varepsilon$ and n_0 is its call depth, $\mathrm{p}(n_0, q)$ lets us conclude $t \in \mathrm{dsem}^{(n)}(q) \leq \mathrm{dsem}(q)$ ($\Leftarrow$) by item (3).

4. If $\perp \in \mathrm{dsem}(q)$, then $\perp \in \mathrm{dsem}^{(n)}(q)$ for all $n \in \mathbb{N}$, so there are derivations of arbitrary call depth or containing infinitely many iterations by (4). In the second case the execution is infinite, and in the first one the execution tree is infinite and by Corollary B.1 there is an infinite execution too.

5. If otherwise $\perp \notin \mathrm{dsem}(q)$ then $\perp \notin \mathrm{dsem}^{(n_0)}(q)$ for some $n_0 \in \mathbb{N}$ by the form of the supremum, then all executions for q are finite.

□

Theorem 3.2 (page 69). *For all $t, t' \in T_\Sigma$, strategy expression α and $\theta \in$ VEnv,*

$$t' \in [\![\alpha]\!](\theta, t) \quad\iff\quad t \mathbin{@} \alpha\, \theta \to^*_{s,c} t' \mathbin{@} \varepsilon$$

and $\perp \in [\![\alpha]\!](\theta, t)$ iff there is an infinite derivation from $t \mathbin{@} \alpha\, \theta$.

Proof. It follows immediately from the previous proposition, by taking $q \coloneqq t_0 \mathbin{@} \alpha\, \theta$. □

Proposition 3.2 (page 74). *For any ∞-recursively enumerable language $L \subseteq \Gamma_\mathcal{R}$, there is some strategy expression α such that $E(\alpha) = L$.*

Proof. The finite-word part L_* and the infinite-word part L_ω of L can be considered separately. In effect, if there is a strategy expression α such that $E(\alpha) = L_*$ and a strategy expression β such that $E(\beta) = L_\omega$, then $E(\alpha \mid \beta) = E(\alpha) \cup E(\beta) = L_* \cup L_\omega = L$.

Let us start with the finite-word part L_*. Since it is recursively enumerable, there must be a Turing machine $M = (Q, \Gamma, T_\Sigma, q_0, F, \delta)$ such that $L_* = L(M)$. Turing machines can easily be represented in Maude, but for the sake of brevity we will see them as terms with two defined operators: `accept` that evaluates to `true` if the word in its tape is accepted, and `append` that puts a symbol on its tape. A generic specification including these functions is available in the example collection. The strategy that admits exactly L_* is defined as a recursive expression that carries a Turing machine as an argument and fills the tape with the visited terms while rewriting. At some point, it runs the Turing machine to decide if the accumulated word is accepted and can be yielded as a solution of the strategy.

```
sd climb(M, N) := run(M, N) | climb(M, s(N)) .
sd run(M, 0) := match S s.t. accept(append(M, S)) .
sd run(M, s(N)) := matchrew S by S using (all ; run(append(M, S), N)) .
```

The initial strategy call is `climb(M0, 0)` where `M0` is in its initial state with an empty tape. Observe that this nonterminating strategy `climb` fixes in advance the length of the executions to be recognized by `run`. This is a technical detail to avoid admitting infinite executions that are accumulation points of the finite words in the language. We claim that `run(M0,` n`)` admits all words in L_* of length n. In effect, the contents of the tape of `M` are the sequence of terms visited until but not including the current subject term `S`. If the second argument is positive, the current state is appended to the tape by `append(M, S)`, a new rewrite step is performed, hence maintaining the invariant in the previous sentence, and `run` is called with $n - 1$. If the counter is zero, the Turing machine `M` is executed by `accept(M')` after appending the last state `S`, which only evaluates to `true` if the word or execution in its tape is in L_*, and only in this case the strategy yields a solution. Finally, the strategy `climb` clearly admits the union of all executions allowed by `run(M,` n`)` for all $n \in \mathbb{N}$, which are all the bounded subsets of L_*, so it admits L_*. Moreover, it does not admit any other word since the infinite $\to_{s,c}$-execution repeating

$$t \mathbin{@} \texttt{climb(M, } n\texttt{)} \to_c t \mathbin{@} (\texttt{run(M, } n\texttt{) | climb(M, } n+1\texttt{)}) \to_c t \mathbin{@} \texttt{climb(M, } n+1\texttt{)}$$

does not contain a single system transition or $\twoheadrightarrow$ step. Naively, we could have defined the strategy as simply

```
sd run(M) := matchrew S s.t. M' := append(M, S) by S using (
  match S s.t. accept(M') | all ; run(M')
) .
```

However, while representing the language $L_* = \{t\}^*$ for some term t, the infinite repetition of t will be inevitably allowed because of the execution that always takes the second branch.

The case of ω-languages is more complicated, but the proof is similar. We assume that the language L_ω is represented by a nondeterministic Turing machine with Büchi conditions and type 2 semantics [Fin14a]. They are defined as tuples $M = (Q, \Sigma, \Gamma, \delta, F, q_0)$ where Q is a finite set of states, Σ is a finite input alphabet (in our case, a subset of T_Σ), Γ is a finite tape alphabet with $\Sigma \subseteq \Gamma$, F is a set of states to define the Büchi condition, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, S\})$ is the nondeterministic transition function. A *run* of M for a word w is an infinite sequence of configurations $r = (q_i, w_i, j_i)_{i=1}^\infty$ with $r_1 = (q_0, w, 0)$ and $r_{k+1} = (q_{k+1}, w_k[j_k/s], j_k + m)$ if $(q_{k+1}, s, m) \in \delta(q_k, (w_k)_{j_k})$ where m is -1 for L, 1 for R, and 0 for S. A run is *complete* if every position of the tape is eventually visited. A word is *accepted* if there is a run such that $q_i \in F$ infinitely often.

```
sd climb(M, N) := run(M, N) | climb(M, s(N)) .
sd run(M, 0) := match S s.t. final(M) ; climb(M, 1) .
sd run(M, s(N)) := match S s.t. needsInput(M) ?
    all ; matchrew S' by S' using run(put(M, S'), s(N))
  : matchrew S s.t. M', MS := step(M) by S using run(M', N) .
```

The `climb` definition is identical to the finite case, but now it fixes the next configuration where a final state of the Turing machine must be found, and it is called repeatedly to ensure that those are visited infinitely often. Since executing the Turing machine after writing an infinite word into the tape is not possible, we advance it while running the strategy and fill the tape lazily with `put` when required. The `step` operator calculates all possible next configurations for the ω-Turing machine, as a comma-separated set. When the machine moves right to a blank position, this is revealed by the `needsInput` predicate, a rewrite step is executed, and the new term is put in place of the blank before it can be read. Each step of the Turing machine consumes the counter and when it bumps into zero, the current state of the Turing machine is checked to be final. If it is not, the execution is discarded. Otherwise, a new call to `climb` ensures that a final state will be visited again.

Let α be `climb(M0, 0)`. First, $E(\alpha) \subseteq L(M)$. If $w \in E(\alpha)$, by definition of $E(\alpha)$ and $\twoheadrightarrow$, there must be some $(q_n)_{k=0}^\infty \in \mathcal{XS}^\omega$ and $(n_k)_{k=0}^\infty \in \mathbb{N}^\omega$ such that $q_n \to_{s,c} q_{n+1}$, $q_{n_k} \twoheadrightarrow q_{n_{k+1}}$ and $\mathrm{cterm}(q_{n_k}) = w_k$. The only rule application in the strategies involved is the **`all`** in the positive branch of the conditional of the second `run` definition. Hence, this branch must have been executed infinitely many times, and so the machine must have moved its head infinitely many times to positions of the tape that need input. The machine is moved only in the negative branch of the same definition by a strategy-call $\to_c$ transition, so let $(m_k)_{k=0}^\infty$ be the indices of the states followed by these transitions. Taking the argument `M` of these calls, a run $(c_k)_{k=0}^\infty$ of the Turing machine can be constructed. In effect, $c_k \vdash c_{k+1}$ by the meaning of `step`, the run is complete since it visits infinitely many positions of the tape, and so the entire tape since it moves only one cell at a time, and the contents of the tape are the word w since this is what `put` inserts each time a rule is executed. Moreover, the Büchi condition is satisfied because of the `climb` strategy: at any configuration c_k, the number of steps until a new final state is reached can be read from the second argument of the `run` call. In conclusion, $(c_k)_{k=0}^\infty$ is an accepting run of the machine, and so $(w_k)_{k=0}^\infty \in L(M)$.

To prove the converse $L(M) \subseteq E(\alpha)$, let $(c_k)_{k=0}^\infty$ be a complete and accepting run of the Turing machine for some word $w \in L_\omega$. Since it is accepting, it must visit infinitely many final states, and there exists $(n_k)_{k=0}^\infty$ with $n_k \geq 1$ such that c_{n_k} is in a final state. Moreover, since the run is complete, all the positions of the tape must be visited, so there is a $(m_k)_{k=0}^\infty$ such that the machine visits the position k for the first time in c_{m_k}. With these ingredients, we can construct a nonterminating derivation of the operational semantics: starting at $q_0 = t_0 \mathbin{@} \alpha$, `climb` calls `run` with n_0, and then the execution of `run` is deterministic until `N` reaches zero except for **`all`** and the selection of the move of the nondeterministic machine. Each time the second branch of the conditional has to be executed, we choose the next machine configuration c_k in the run in the **`matchrew`**. Similarly, when the first branch is executed, the result of **`all`** is chosen to match the value of the current cell in c_k and this is always possible since $(w_k)_{k=0}^\infty$ is a valid rewriting path of the uncontrolled system. When the counter descends to zero, the test in the `run` definition is satisfied, since the configuration is some final c_{n_k}, and the new `run` argument generated by `climb` is chosen to be $n_k - n_{k+1}$, and this procedure is repeated forever. The resulting $\to_{s,c}$ derivation contains infinitely many $\to_s$ transitions as a consequence of the completeness of the machine run, and so a $\twoheadrightarrow$ derivation can be extracted whose projection is the expected word w since the `all` outputs have been chosen to match it. Therefore, $w \in E(\alpha)$.

□

Proposition 3.3 (page 74). *If the reachable states from $t \mathbin{@} \alpha$ are finitely many, $E(\alpha, t)$ is a closed ∞-regular language.*

Proof. The Büchi automaton for $E(\alpha, t)$ is $A = (Q, \mathrm{cterm}(Q), \delta, \{start\}, Q)$ where $Q = \{start\} \cup \{q \in \mathcal{XS} \mid t \mathbin{@} \alpha \twoheadrightarrow^* q\} \cup \{t' \mathbin{@} \varepsilon : t \mathbin{@} \alpha \to_{s,c}^* t' \mathbin{@} \varepsilon\}$ and

$$\begin{array}{rll}
\delta(start, t) = & \{\ t \mathbin{@} \alpha\ \} & \\
\delta(start, t') = & \varnothing & \text{if } t' \neq t \\
\delta(q, t') = & \{\ q' : q \twoheadrightarrow q' \ \wedge\ \mathrm{cterm}(q') = t'\ \} & \text{for } q \in Q \setminus \{start\} \\
& \cup\ \{\ t' \mathbin{@} \varepsilon : \text{if } q \to_c^* t' \mathbin{@} \varepsilon\ \} &
\end{array}$$

The identity $L(A) = E(\alpha, t)$ follows from the fact that runs π in A yield executions $\mathrm{cterm}(\pi)$ in $E(\alpha, t)$ and vice versa. Proving this is straightforward, on account of the definitions of A and $E(\alpha, t)$. The proof for finite words is identical.

□

Lemma B.10. *Given two terms $t, t' \in T_\Sigma$ such that $t \to_R^1 t'$, there is a strategy $\alpha_{t,t'}$ of the form*

$$\texttt{matchrew}\ P\ \texttt{by}\ x\ \texttt{using}\ rl[\rho]\{\overline{\beta}\}\ ;\ \texttt{match}\ t'$$

such that $t \mathbin{@} \alpha_{t,t'} \to_{s,c}^* u \mathbin{@} \varepsilon$ iff $t' = u$ and there are finitely many reachable states from $\alpha_{t,t'}$.

Proof. Notice that the much simpler strategy `all ; match` t' also satisfies the first requirement, but not necessarily the second since the rewriting condition may have infinitely many solutions. If $t \to_R^1 t'$, there must exist a (perhaps conditional) rule $rl : l \to r$ if C, a substitution σ, and a position p in t such that $t|_p = \sigma(l)$, $t' = t[p/\sigma(r)]$ and $\sigma(C)$ holds. Proceeding by induction on the number of rewriting conditions required to prove a step, we first suppose that C does not contain rewriting condition fragments. If $x_1, \ldots, x_n$ are the variables that occur in l and C, and x is a fresh variable, the desired α is then

$$\texttt{matchrew}\ t[p/x]\ \texttt{by}\ x\ \texttt{using}\ rl[x_1 \leftarrow \sigma(x_1), \ldots, x_n \leftarrow \sigma(x_n)]\ ;\ \texttt{match}\ t'$$

This strategy forces the application of a rule with label rl to the specific position p, with the specific substitution σ. This does not always guarantee that the only possible rewrite is $t \to_R^1 t'$, because there may be multiple rules with the same label, but the final test does.

$$\begin{array}{l}
t \mathbin{@} \alpha_{t,t'} \to_c \mathrm{subterm}(x : t|_p \mathbin{@} rl[x_k \leftarrow \sigma(x_k)]_{k=1}^n ; t[p/x]) \mathbin{@} \texttt{match}\ t' \\
\qquad \to_s \mathrm{subterm}(x : \sigma(r) \mathbin{@} \varepsilon ; t[p/x]) \mathbin{@} \texttt{match}\ t' \to_c t_{rl} \mathbin{@} \texttt{match}\ t' \to_c t' \mathbin{@} \varepsilon
\end{array}$$

Every possible execution from $t \mathbin{@} \alpha_{t,t'}$ is like the one above, whose last step only holds if $t_{rl} = t'$.

If C contains rewriting conditions, we have to indicate strategies for these. Since $t \to_R^1 t'$, for each rewriting condition $l' \Rightarrow r'$, a sequence $t_1 t_2 \cdots t_n$ must exist with $\sigma(l') = t_1$, $\sigma(r') = t_n$ and $t_k \to_R^1 t_{k+1}$. Some of these steps may apply rules with rewriting conditions, but we are one level less, so the existence of $\alpha_{t_k, t_{k+1}}$ can be assumed. Joining all the transitions with the concatenation operator of strategies, a strategy is built to solve one of the rewriting conditions. The same can be done for the other rewriting fragments, so the lemma holds. □

Proposition 3.4 (page 74). *If L is a closed ∞-regular language, there is a strategy expression β such that $E(\beta) = L$ and the reachable states from $t \mathbin{@} \beta$ are finitely many for all $t \in T_\Sigma$.*

Proof. The proofs for the finite and the infinite cases are similar, so only the infinite case is considered. Approximately, the strategy expression β will be the translation of the ω-regular expression for the language L. Since L is ω-regular, there must be a Büchi automaton $A = (Q, S, \delta, Q_0, F)$ for L. However, the symbols of the alphabet are states and our language is based on rules, so we have to translate it. The translation $B = (S{\times}Q, RA, \Delta, B_0, S{\times}F)$ is defined using the strategies of Lemma B.10, $RA = \{\alpha_{t,t'} \mid t, t' \in S\}$ with $\Delta((t,q),\alpha) = \{\ (t',q') \mid t \mathbin{@} \alpha \to^*_{s,c} t' \mathbin{@} \varepsilon,\ q' \in \delta(q,t')\ \}$ and $B_0 = \{\ (w_0, q) \mid w \in L,\ q \in \delta(q_0, t),\ q_0 \in Q_0\ \}$. It is easy to prove that $\alpha_{t,t'}$ satisfies the definition of Δ for any pair of terms and that $L = \{v \in T^\omega_\Sigma \mid v_k \mathbin{@} w_k \to^*_{s,c} v_{k+1} \mathbin{@} \varepsilon$ for all $k \in \mathbb{N}, w \in L(B)\}$.

Since $L(B)$ is ω-regular, it can be expressed as an ω-regular expression [LT16], which always have the form $r_1 s_1^\omega + \ldots + r_n s_n^\omega$ for r_i, s_i regular expressions and $\varepsilon \notin L(s_i)$. The conversion from regular expressions to strategy expressions is almost an identity. $\varnothing$ is translated as **`fail`**, ε as **`idle`**, alternation, concatenation, and iterations are the same in both languages. For each s_i, to represent s_i^ω, we define a named strategy with label f_i without argument and defined as $T(s_i) ; f_i$ if T is the translation function.

Inductively, we will prove that any successful execution $t \mathbin{@} T(r)$ for any regular or ω-regular expression r sequentially executes all strategies in a word $w \in L(r)$, and that all words in $L(r)$ can be successfully executed for some initial term. This implies, from what we have proved above, that the traces for $T(r)$ are exactly L as we want to prove. Splitting the execution in RA *atoms* is always possible, since they are the only rule applications in the sequence enclosed by **`matchrew`** opening and closing transitions, with possibly some control steps between these atoms. The proof is by induction on the structure of regular and ω-regular expressions. However, we should take care that the semantics of the iteration is different from that of the Kleene star, since the iteration body could be repeated indefinitely. Since $E(\alpha, t)$ is closed, this infinite execution will be already included, so it makes no difference.

Finally, and since the strategy satisfies the conditions of the second statement of Proposition 3.5, the reachable states are finite. Notice that Proposition 3.5 appears after the current proposition in Section 3.6 and so in this appendix, but this is just for presentation purposes. There are no cyclic dependencies between the two results, as follows from the self-contained proof of Proposition 3.5.

□

Proposition 3.5 (page 75). *The reachable states from $t \mathbin{@} \alpha$ are finitely many if any of the following conditions holds:*

1. *α does not contain iterations or recursive calls.*

2. *The reachable terms from $t \mathbin{@} \alpha$ are finitely many and all recursive calls in α and the reachable strategy definitions are tail.*

Proof. In the first statement, no recursive calls are allowed, but non-recursive calls are admissible. This means that the static call graph is an acyclic graph for which a topological ordering can be obtained. The original strategy α can also be inserted in that order and the proof can be accomplished by an induction on the topological ordering followed by an induction on the execution states ranked by the lexicographic combination of the number of strategy constructors in their stacks, the number of execution state constructors, and the number of condition fragments in rewc states. Looking at the rules, each possible execution state has a finite number of successors by the $\to_{s,c}$ relation, and the induction hypothesis can be applied for all but iterations and calls.

For the second statement, we know that the number of terms that can appear either as subject or as strategy call arguments in execution states is finite, so iteration and tail-recursive calls can be handled. The body α of an iteration $\alpha{*}$ cannot contain recursive strategy calls, because it would not be tail calls. Inductively on the number of nested iterations, there are finitely many reachable states from $t \mathbin{@} \alpha{*}\, s$ in addition to those from the reachable $t' \mathbin{@} s$ states at the end of the iteration. The only direct successors of $t \,@\, \alpha{*}\, s$ are $t \,@\, \alpha\, \alpha{*}\, s$ and $t \,@\, s$. The reachable states from $t \,@\, \alpha\, \alpha{*}\, s$ are those reachable from $t \,@\, \alpha\, \mathrm{vctx}(s)$ with $\mathrm{vctx}(s)$ replaced by $\alpha{*}\, s$, and those reachable from $t' \,@\, \alpha{*}\, s$ for each solution yielded by $t \,@\, \alpha\, \mathrm{vctx}(s)$. The former are finitely many if α does not contain iterations by the first statement. Otherwise, they are a finite amount by induction hypothesis, since any iteration β^* inside α contains fewer nested iterations, and so its reachable states are finitely many. Extending this property to the whole α is a straightforward structural induction identical to the proof of the first statement. Considering the successors of $t' \,@\, \alpha{*}\, s$ for all reachable t' together, we only have to sum the other reachable states we have mentioned for each t', which yields a finite sum of finite quantities. Hence, the reachable states before s are finitely many.

Consider now an execution state $t \,@\, sl\,(t_1, \ldots, t_n)\, \sigma\, s$ where sl is a recursive strategy. Its successors are $t \,@\, \delta\, \sigma'\, s$ for some definition δ and substitution σ'. If the expression δ does not contain recursive strategies, finitely many states are reachable from $t \,@\, \delta\, \sigma'$. Otherwise, all recursive calls are tail and this yields finitely many states plus some $t_k \,@\, sl_k(t_{k,1}, \ldots, t_{k,n_k})\, \sigma'\, s$ and their successors. More precisely, it should be proved that our syntactical definition of tail call ensures this, but it is a straightforward inductive check. Since the possible t_k, $t_{k,l}$, and sl_k are finitely many, the successors of the initial state before reducing s are finitely many, and combining all the results the whole reachable states are a finite set. □

B.2 Model checking strategy-controlled systems

Proposition 4.1 (page 86). *Given a linear-time property φ, $(\mathcal{K}, E(\lambda)) \vDash \varphi$ iff $(\mathcal{K}, E(\lambda)) \vDash' \varphi$.*

Proof. The executions of the unwinding once projected coincide with the executions in $E(\lambda)$. □

Lemma B.11. *Every closed ω-language is recognized by a deterministic Büchi automaton with trivial acceptance conditions.*

Proof. This statement is in [PP04; Proposition 3.7]. We have also proved it in [RMP+19b; Lemma 2] without the deterministic adjective, which can be easily added by the powerset construction used for finite automata, since the obstacle that impedes determinizing arbitrary Büchi automata are the Büchi conditions. □

Lemma B.12. *If $\mathcal{K}$ and $\mathcal{K}'$ are bisimilar, $\bigcup_{s \in I} \ell(\Gamma_{\mathcal{K},s}) = \bigcup_{s \in I'} \ell'(\Gamma_{\mathcal{K}',s})$.*

Proof. $\mathcal{K}$ and $\mathcal{K}'$ are interchangeable in the lemma, so it is enough to prove $G \subseteq G'$ by induction. If G and G' are the unions in the statement, this is solved by induction with the property $p(ws) = \exists i' \in I', w's' \in \Gamma_{\mathcal{K}',i'} \quad \ell(ws) = \ell'(w's') \wedge (s, s') \in B$ for all $i \in I$ and $ws \in \Gamma_{\mathcal{K},i}$, where B is a bisimulation. The infinite executions are the limits of the finite ones, so they coincide too. □

Proposition 4.2 (page 87). *Given a Kripke structure $\mathcal{K}$ and a closed strategy $E = E(\lambda) \subseteq (S \cup A)^\omega$,*

$$(1) \Rightarrow (2) \Rightarrow (3) \Leftrightarrow (4)$$

1. *E is ω-regular.*
2. *There is a finite Kripke structure $\mathcal{K}'$ bisimilar to $\mathcal{U}(\mathcal{K}, \lambda)$.*
3. *$\ell(E)$ is ω-regular.*
4. *There is a finite Kripke structure $\mathcal{K}'$ such that $\ell'(\Gamma^\omega_{\mathcal{K}'}) = \ell(E)$.*

Proof. The basic idea all along the proof is that Büchi automata can be used with a few changes as Kripke structures and vice versa. These manipulations are simple, but transition labels make them more obscure.

(1) ⇒ (2). If $E(\lambda)$ is ω-regular and closed, it is recognized by a deterministic Büchi automaton $M = (Q, S \cup A, \delta, \iota, Q)$ with trivial conditions by Lemma B.11. Moreover, all words accepted by this automaton alternate states in S with actions in A. Let $\mathcal{K}'$ be $(Q \times S, R', I', AP, \ell \circ \pi_2)$ where $I' = \{(q, s) : q \in \delta(\iota, s), s \in I\}$, π_2 is the second projection of the pair, and $((q, s), a, (q', s')) \in R' \iff \exists\, q_m \in Q \quad q_m \in \delta(q, a) \wedge q' \in \delta(q_m, s')$. $\mathcal{U}(\mathcal{K}, \lambda)$ is bisimilar to $\mathcal{K}'$ by the following relation $B = \{(ws, (q, s)) : q \in \hat{\delta}(\iota, ws)\}$ where $\hat{\delta}(q, \varepsilon) = \{q\}$ and $\hat{\delta}(q, wx) = \bigcup_{q' \in \hat{\delta}(q,w)} \hat{\delta}(q', x)$. First, it is clear that states related by B have the same label. Second, $\mathcal{K}'$ simulates $\mathcal{U}(\mathcal{K}, \lambda)$. In effect, assuming $ws \to wsas'$ and $(ws, (q, s)) \in B$, there is a state (q', s') where $q' \in \hat{\delta}(\iota, wsas')$ that satisfies $(wsas', (q', s')) \in B$ by definition and $((q, s), a, (q', s')) \in R'$. $\hat{\delta}(\iota, wsas')$ is not empty because $wsas'$ is a prefix of an execution allowed by the strategy, and so recognized by M. The transition is in R' by definition of $\hat{\delta}$, since q' satisfies $q' \in \delta(q_m, s')$ with $q_m \in \delta(q_0, a)$ for some $q_0 \in \hat{\delta}(\iota, ws)$, but $q_0 = q$ because of the determinism of M. The other simulation is proven similarly.

(2) ⇏ (1). For an unlabeled counterexample, take $\mathcal{K} = (\{a, b\}, \{a, b\}^2, \{a\}, \varnothing, \varnothing)$, $\lambda(w) = \{a, b\}$ if $w = a^n$ for n prime and $\{a\}$ otherwise, and $\mathcal{K}' = (\{c\}, \{(c, c)\}, \{c\}, \varnothing, \varnothing)$. $E(\lambda)$ is not ω-regular and $\mathcal{K}'$ is bisimilar to $\mathcal{U}(\mathcal{K}, \lambda)$ by the only possible total relation. Hence, we only prove that $\ell(E(\lambda))$ is ω-regular.

(2) ⇒ (3). Given $\mathcal{K}' = (S', R', I', AP, \ell')$, we define the ω-automaton $M = (Q, \mathcal{P}(AP) \cup A, \delta, \iota, Q)$ where $Q = S' \times \{0, 1\} \cup \{\iota\}$ and $\delta(\iota, P) = \{(s, 1) : \ell(s) = P, s \in I'\}$, $\delta((s, 1), a) = \{(s', 0) : (s, a, s') \in R'\}$ and $\delta((s, 0), P) = \{(s, 1)\}$ if $P = \ell(s)$. It is clear that the runs of M are of the form $\iota\,(s_0, 1)(s_1, 0)(s_1, 1)(s_2, 0)(s_2, 1) \cdots$ and they accept words $s_0 a_1 s_1 a_2 s_2 \cdots$ that coincide with the executions of $\mathcal{K}'$. By Lemma B.12, the projected executions of $\mathcal{K}'$ coincide with those of $\mathcal{U}(\mathcal{K}, \lambda)$ and these are exactly $\ell(E(\lambda))$. Hence, M is a Büchi automaton for the ω-regular language $\ell(E(\lambda))$. Finally, $\ell(E(\lambda))$ being ω-regular is not enough for the existence of a bisimilar finite $\mathcal{K}'$. The previous counterexample can be refined to show this.

(3) ⇔ (4). If $\ell(E)$ is ω-regular, there must be an automaton $M = (Q, \mathcal{P}(AP) \cup A, \delta, \iota, Q)$, with trivial Büchi conditions since $\ell(E)$ is closed too. We can define the finite Kripke structure $\mathcal{K}' = (Q \times \mathcal{P}(AP), R', I', AP, \pi_2)$ with $I' = \{(q, P) : q \in \delta(\iota, P), P \in \mathcal{P}(AP)\}$ and $((q, P), a, (q', P')) \in R'$ if $\exists\, q_m \in Q \quad q_m \in \delta(q, a) \wedge q' \in \delta(q_m, P')$. Every execution π in $\mathcal{K}'$ combines an arbitrary run in M in the first component with the word accepted by this run in the second. The labeling function $\ell' = \pi_2$ projects this second component. Since all runs in M are accepting, all projections $\pi_2(\pi)$ are exactly $\ell(E)$ with the runs $\pi_1(\pi)$ as a proof. Conversely, if there is a finite $\mathcal{K}' = (S', R', I', AP, \ell')$ such that $\ell'(\Gamma^{\omega}_{K'}) = \ell(E)$, we only have to see the Kripke structure as an automaton, like in (2) ⇒ (3). □

Proposition 4.3 (page 88). *Given a CTL* formula* Φ, $\mathcal{K}, s \vDash \Phi$ *iff* $\Gamma^{\omega}_{\mathcal{K},s} \vDash \Phi$.

Proof. The key fact is that the possible continuations of any finite execution ws for $\Gamma^{\omega}_{\mathcal{K}}$ only depend on its final state s, since the executions are unrestricted. Hence, $\Gamma^{\omega}_{\mathcal{K},s} \upharpoonright (ws') = \Gamma^{\omega}_{\mathcal{K},s'}$ for all $ws' \in \Gamma^{*}_{\mathcal{K},s}$.Then, $\mathcal{K}, s \vDash \Phi$ iff $\Gamma^{\omega}_{\mathcal{K},s} \vDash \Phi$ and $\mathcal{K}, s \vDash \phi$ iff $\Gamma^{\omega}_{\mathcal{K},\pi_0}, \pi \vDash \phi$ can be easily proven by induction on the formula. Almost syntactically, $\Gamma^{\omega}_{\mathcal{K},s}$ can be replaced by s and the strategy can be removed in the path relation to obtain the classical definition. The two properties clearly hold in 1 to 7. In 8 and 9, with $E = \Gamma^{\omega}_{\mathcal{K},\pi_0}$, we can observe that $E \upharpoonright \pi_0 \pi_1 = \Gamma^{\omega}_{\mathcal{K},\pi_1}$ and $\pi_1 = (\pi^1)_0$, and that $E \upharpoonright \pi^{\leq n} = \Gamma^{\omega}_{\mathcal{K},\pi_n}$ with $\pi_n = (\pi^n)_0$. The induction hypothesis can then be applied. □

Lemma B.13. *For every* $ws_0 \in S^+$ *prefix in* $E(\lambda)$, $E(\lambda) \upharpoonright ws_0 = \{\mathrm{flat}(\pi) : \pi \in \Gamma_{\mathcal{U}(\mathcal{K},\lambda),ws_0}\}$ *where* $\mathrm{flat}((ws_0)(ws_0 s_1)(ws_0 s_1 s_2) \cdots) \coloneqq s_0 s_1 s_2 \cdots$.

Proof. Executions in $\mathcal{U}(\mathcal{K},\lambda)$ are of the form $(ws_0)(ws_0s_1)(ws_0s_1s_2)\cdots$ where $s_0s_1\cdots$ is an execution in E. For the $\supseteq$ inclusion, take $\Gamma_{\mathcal{U}(\mathcal{K},\lambda),ws_0} \ni \pi = (ws_0)(ws_0s_1)(ws_0s_1s_2)\cdots$, whose $\text{flat}(\pi) = s_0s_1s_2\cdots$ and $ws_0s_1s_2\cdots \in E$ since $s_{n+1} \in \lambda(s_n)$. Hence, $\text{flat}(\pi) = s_0s_1\cdots \in E \upharpoonright ws_0$ by definition. For the other $\subseteq$ inclusion, $s_0s_1\cdots \in E \upharpoonright ws_0$ implies $ws_0s_1\cdots \in E$, so $\pi = (ws_0)(ws_0s_1)\cdots \in \Gamma_{\mathcal{U}(\mathcal{K},\lambda),ws_0}$ and $\text{flat}(\pi) = s_0s_1\cdots$ is in the set. □

Proposition 4.4 (page 88). *Given $(\mathcal{K}, E(\lambda))$ and a CTL* formula φ, $\mathcal{U}(\mathcal{K},\lambda) \vDash \varphi$ iff $\mathcal{K}, E(\lambda) \vDash \varphi$.*

Proof. We follow an inductive proof on the structure of CTL* formulae with the more general property $\mathcal{U}(\mathcal{K},\lambda), w \vDash \varphi$ iff $\mathcal{K}, E(\lambda) \upharpoonright w \vDash \varphi$ for all $w \in S^+$. Path formulae need to be handled simultaneously, so the inductive property also includes $\mathcal{U}(\mathcal{K},\lambda), \pi \vDash \varphi$ iff $\mathcal{K}, E(\lambda) \upharpoonright \pi_0, \text{flat}(\pi) \vDash \varphi$ (in the lefthand side executions are successions of growing S^+ words while in the righthand side they are successions of S states). To facilitate reading, we will omit the $\mathcal{U}(\mathcal{K},\lambda)$ and $\mathcal{K}$ prefix when writing the satisfaction relations, and use E for $E(\lambda)$.

- (p, atomic propositions) By definition, $ws \vDash p$ iff $p \in \ell(s)$, and $E \upharpoonright ws \vDash p$ iff $p \in \ell(s')$ for all $s'w' \in E \upharpoonright ws = \{sw'' : wsw'' \in E\}$. Then, s' can only be s and both conditions coincide.

- ($\Phi_1 \wedge \Phi_2$) In the standard side, the conjunction is satisfied iff $w \vDash \Phi_i$ for both $i = 1, 2$. In the strategy side, this happens iff $E \upharpoonright w \vDash \Phi_i$. By induction hypothesis on both Φ_i the equivalence holds.

- ($\neg\Phi$) The same inductive argument can be used for negation.

- ($\mathbf{A}\,\varphi$) This formula is satisfied iff $\pi \vDash \varphi$ for all $\pi \in \Gamma^{\omega}_{\mathcal{U}(\mathcal{K},\lambda),w}$ in the $\mathcal{U}(\mathcal{K},\lambda)$ side. In the strategy side, this is $E \upharpoonright w, \rho \vDash \varphi$ for all $\rho \in E \upharpoonright w$. Using Lemma B.13, all these ρ are exactly those $\text{flat}(\pi)$, and applying the induction hypothesis on φ, both statements are equivalent.

Let π be $(ws_0)(ws_0s_1)\cdots$, we then target the path satisfaction cases:

- ($\bigcirc\,\varphi$) We should prove that $\pi \vDash \bigcirc\,\varphi$ is equivalent to $E \upharpoonright ws_0, s_0s_1\cdots \vDash \bigcirc\,\varphi$. Their definitions translate these to $\pi^1 \vDash \varphi$ and $(E \upharpoonright ws_0) \upharpoonright s_0s_1, (s_0s_1\cdots)^1 \vDash \varphi$. But they are equivalent by induction hypothesis on φ, since $(E \upharpoonright ws_0) \upharpoonright s_0s_1 = E \upharpoonright ws_0s_1 = E \upharpoonright \pi_1 = E \upharpoonright (\pi^1)_0$ and $(s_0s_1\cdots)^1 = s_1s_2\cdots = \text{flat}(\pi^1)$.

- ($\varphi_1\,\mathbf{U}\,\varphi_2$) The formula holds in the standard sense if there is an $n \in \mathbb{N}$ such that $\pi^n \vDash \varphi_2$ and for all k such that $0 \leq k < n$ then $\pi^k \vDash \varphi_1$. In the strategy side, the formula holds if again there is an $n \in \mathbb{N}$ such that $(E \upharpoonright ws_0) \upharpoonright s_0\cdots s_n, s_ns_{n+1}\cdots \vDash \varphi_2$ and $(E \upharpoonright ws_0) \upharpoonright s_0\cdots s_k, s_ks_{k+1}\cdots \vDash \varphi_1$ for all $0 \leq k < n$. Since $(E \upharpoonright ws_0) \upharpoonright s_0\cdots s_k = E \upharpoonright ws_0\cdots s_k = E \upharpoonright (\pi^k)_0$ and $s_ks_{k+1}\cdots = \text{flat}(\pi^k)$ for all $k \in \mathbb{N}$, the induction hypothesis can be applied to both φ_1 and φ_2 to conclude the property for $\varphi_1\,\mathbf{U}\,\varphi_2$.

- (Φ) $\pi \vDash \Phi$ is defined as $\pi_0 \vDash \Phi$ in the standard sense, and $E \upharpoonright ws_0, s_0s_1\cdots \vDash \Phi$ is $E \upharpoonright ws_0 \vDash \Phi$ in the strategy case. Since $\pi_0 = ws_0$, both statements are related as in the induction property. The hypothesis on Φ itself can be applied, considering that state satisfaction is below path satisfaction in the induction order (we have never used this argument in reverse), and then they are equivalent.

A complete subset of CTL* constructors has been handled in the proof, the derived operators follow from the well-known semantic equivalences. □

Proposition 4.5 (page 89). *Given $(\mathcal{K}, E)$ and a closed μ-calculus formula φ, $s \in [\![\varphi]\!]_{\mathcal{K},\eta}$ iff $\Gamma_{\mathcal{K},s} \in \langle\!\langle\varphi\rangle\!\rangle_{\mathcal{K},\xi}$ for any η and ξ.*

Proof. This property can be proven inductively, adding the variable valuations to the inductive property and the premise that $\eta(Z) \ni s$ iff $\Gamma_{\mathcal{K},s} \in \xi(Z)$ for all variables Z. For the initial φ, this premise is trivially satisfied since we can take $\eta(Z) = \varnothing = \xi(Z)$ regardless of the given two, since the formula is closed. We will not detail some trivial cases:

- (p) By definition, $s \in [\![p]\!]_\eta$ is $p \in \ell(s)$ and $\Gamma_s \in \langle\!\langle p \rangle\!\rangle_\xi$ is $\forall \pi \in \Gamma_s\ p \in \ell(\pi_0)$. Since Γ_s are the executions of $\mathcal{K}$ starting at s, $\pi_0 = s$ and both statements are equivalent.
- ($\langle a \rangle \varphi$) $s \in [\![\langle a \rangle \varphi]\!]_\eta$ if there is an $s' \in S$ such that $s \to^a s'$ and $s' \in [\![\varphi]\!]_\eta$. On the other side, $\Gamma_s \in \langle\!\langle \langle a \rangle \varphi \rangle\!\rangle_\xi$ holds iff there is $saw \in \Gamma_s$ such that $\Gamma_s \upharpoonright saw_0 = \Gamma_{w_0} \in \langle\!\langle \varphi \rangle\!\rangle_\xi$. The induction hypothesis with $s' = w_0$ lets us conclude the property.
- ($\nu Z.\varphi$) $s \in [\![\nu Z.\varphi]\!]_\eta$ iff there is a set V such that $s \in V$ and $V \subseteq [\![\varphi]\!]_{\eta[Z/V]}$. In the strategy side, $\Gamma_s \in \langle\!\langle \nu Z.\varphi \rangle\!\rangle_\xi$ iff there is an F such that $\Gamma_s \in F$ and $F \subseteq \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]}$. Assuming there exists a V with these properties ($\Rightarrow$), consider $F = \{\Gamma_s : s \in V\}$. In other words, $s \in V$ iff $\Gamma_s \in F$, so $\eta[Z/V]$ and $\xi[Z/F]$ are properly related. Hence, by induction hypothesis on φ, $\Gamma_s \in \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]}$ iff $s \in [\![\varphi]\!]_{\eta[Z/V]}$, so $F \subseteq \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]}$ as we wanted to prove. In the opposite direction ($\Leftarrow$), assuming the existence of an F with the mentioned properties, consider $V = \{s \in S : \Gamma_s \in F\}$ and the proof is the same.

$\square$

Proposition 4.6 (page 89). *Given* $(\mathcal{K}, E(\lambda))$ *and a closed μ-calculus formula* φ, $s \in [\![\varphi]\!]_{\mathcal{U}(\mathcal{K},\lambda),\eta}$ *for* $s\pi \in E$ *iff* $E \in \langle\!\langle \varphi \rangle\!\rangle_{\mathcal{K},\xi}$ *for any* η *and* ξ.

Proof. Let us inductively prove the more general property that $[\![\varphi]\!]_\eta \ni w$ iff $E \upharpoonright w \in \langle\!\langle \varphi \rangle\!\rangle_\xi$ provided that $\eta(Z) \ni w$ iff $E \upharpoonright w \in \xi(Z)$ for all variables Z.

- (p) By definition, $ws \in [\![\varphi]\!]_\eta$ iff $p \in \ell(s)$, and $E \upharpoonright ws \in \langle\!\langle \varphi \rangle\!\rangle_\xi$ iff $p \in \ell(\pi_0)$ for all $\pi \in E \upharpoonright ws$. However, π_0 must be s since $E \upharpoonright ws = \{sw' : wsw' \in E\}$, so both sides are equivalent.
- (Z) The value of Z in both contexts is respectively $\eta(Z)$ and $\xi(Z)$, so the property directly follows from the assumption about these two functions.
- ($\varphi_1 \wedge \varphi_2$) The standard definition says $[\![\varphi_1 \wedge \varphi_2]\!]_\eta = [\![\varphi_1]\!]_\eta \cap [\![\varphi_2]\!]_\eta$ and the strategy one is $\langle\!\langle \varphi_1 \wedge \varphi_2 \rangle\!\rangle_\xi = \langle\!\langle \varphi_1 \rangle\!\rangle_\xi \cap \langle\!\langle \varphi_2 \rangle\!\rangle_\xi$. Hence, the property holds by induction hypothesis on both φ_1 and φ_2.
- ($\neg\varphi$) By definition, $[\![\neg\varphi]\!]_\eta = S^+ \setminus [\![\varphi]\!]_\eta$ and $\langle\!\langle \neg\varphi \rangle\!\rangle_\xi = \mathcal{P}(\Gamma_\mathcal{K}) \setminus \langle\!\langle \varphi \rangle\!\rangle_\xi$, so the property holds by induction hypothesis on φ.
- ($\langle a \rangle \varphi$) $ws \in [\![\langle a \rangle \varphi]\!]_\eta$ iff there is an $(a, s') \in \lambda(ws)$ such that $wsas' \in [\![\varphi]\!]_\eta$ according to the standard definition of μ-calculus and the transition relation on $\mathcal{U}(\mathcal{K}, \lambda)$. On the other side, $E \upharpoonright ws \in \langle\!\langle \langle a \rangle \varphi \rangle\!\rangle_\xi$ iff there is a $w' \in (S \cup A)^\infty$ such that $saw' \in E \upharpoonright ws$ and $(E \upharpoonright ws) \upharpoonright saw'_0 = E \upharpoonright wsaw'_0 \in \langle\!\langle \varphi \rangle\!\rangle_\xi$.

  By definition of $E(\lambda)$, there is a $w' \in (S \cup A)^\infty$ such that $wsaw' \in E$ iff $(a, w'_0) \in \lambda(ws)$. Hence, by induction hypothesis on φ and taking $w'_0 = s'$, we conclude that the property holds.
- ($\nu Z.\varphi$) According to the standard definition, $ws \in [\![\nu Z.\varphi]\!]_\eta$ iff there is a $V \subseteq S^+$ such that $V \subseteq [\![\varphi]\!]_{\eta[Z/V]}$ and $ws \in V$. According to our definition for strategies, $E \upharpoonright ws \in \langle\!\langle \nu Z.\varphi \rangle\!\rangle_\xi$ iff there is an $F \subseteq \mathcal{P}(\Gamma_\mathcal{K})$ such that $F \subseteq \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]}$ and $E \upharpoonright ws \in F$. Both implications can be proven like in the previous proposition, but taking $F = \{E \upharpoonright w : w \in V\}$ for a given V, and $V = \{w \in (S \cup A)^+ : E \upharpoonright w \in F\}$ for a given F.

□

Proposition 4.7 (page 90). *The projections of the infinite traces of $\mathcal{M}_{\alpha,t}$ by* $\mathrm{cterm} \circ \pi_1$ *coincide with the stuttering extension of $E(\alpha, t)$.*

Proof. Remember that $\mathcal{M}_{\alpha,t}$ is defined as $(\mathcal{XS} \times \{0\} \cup \mathrm{Sol} \times \{1\}, \twoheadrightarrow_{\mathrm{Sol}})$ where $(q, 0) \twoheadrightarrow_{\mathrm{Sol}} (q', 0)$ if $q \twoheadrightarrow q'$ and $(q, k) \twoheadrightarrow_{\mathrm{Sol}} (q, 1)$ if $q \in \mathrm{Sol}$ for $k = 0, 1$. Consequently, all infinite traces of $\mathcal{O}^{\alpha,t}$ are infinite traces of $\mathcal{M}_{\alpha,t}$ (with a zero in the second component), but these are exactly $\mathrm{Ex}^\omega(\alpha, t)$ by definition. The finite traces $q_1 \cdots q_n$ in $\mathcal{O}$ are finite traces $(q_1, 0) \cdots (q_n, 0)$ in $\mathcal{M}_{\alpha,t}$, but if and only if $q_n \in \mathrm{Sol}$ they can be extended to the infinite traces $(q_1, 0) \cdots (q_n, 0)(q_n, 1) \cdots$. These are the traces in $\mathrm{Ex}^*(\alpha, t)$, whose stuttering-extended projections in $E(\alpha, t)$ are precisely $\mathrm{cterm}(q_1) \cdots \mathrm{cterm}(q_n)\, \mathrm{cterm}(q_n) \cdots$, the projection of the extended executions ending in a halting state. Thus, $\mathrm{cterm}(\pi_1(\Gamma^\omega_{\mathcal{M}}))$ is the stuttering extension of $E(\alpha, t)$, where $\pi_1(x, y) = x$. □

Lemma B.14. *If the underlying equational theory is decidable and the reachable states from $t \mathbin{@} \alpha$ are finitely many, $\twoheadrightarrow$ and $\rightarrow_{s,c}$ are decidable.*

Proof. All the rules defining $\rightarrow_c$ and $\rightarrow_s$ but [else] are decidable, since they only involve immediate term manipulations, matching, substitution application, etc. The [else] rule is decidable on an execution state $t \mathbin{@} \beta \mathbin{?} \gamma : \zeta s$ if the reachable states from $t \mathbin{@} \beta\, \mathrm{vctx}(s)$ are finitely many. However, these are already embedded in the states reachable from the conditional, since $t \mathbin{@} \beta \mathbin{?} \gamma : \zeta s \rightarrow_c t \mathbin{@} \beta \gamma s$ and all the successors of $t \mathbin{@} \beta\, \mathrm{vctx}(s)$ are successors of $t \mathbin{@} \beta \gamma s$ with $\mathrm{vctx}(s)$ replaced by γs, since the same rules can be applied with their free variables s changed like this. In case of nested conditionals, this argument can be repeated from inside out conditional expressions, so all these states are reachable from the initial $t \mathbin{@} \alpha$, and so they are finitely many and the rule is decidable. Since the reachable states are a finite set, deciding $q \twoheadrightarrow q'$ is finding a path via $\rightarrow_c$ transitions from q to any predecessor of q' by a $\rightarrow_s$ transition, so it is decidable. □

Theorem 4.1 (page 93). *$\mathcal{M}'_{\alpha,t}$ and $\mathcal{U}(\mathcal{M}, \lambda_{E(\alpha)})$ are bisimilar Kripke structures.*

Proof. Let $f : (T_\Sigma \cup A)^+ \rightarrow \mathcal{P}(\mathcal{XS})$ be defined by $f(t_0 a_1 t_1 \cdots a_n t_n) = \{q_n \in \mathcal{XS} : t_0 \mathbin{@} \alpha = q_0 \twoheadrightarrow^{a_1} \cdots \twoheadrightarrow^{a_n} q_n, \mathrm{cterm}(q_k) = t_k\}$. For any $w \in (T_\Sigma \cup A)^+$ and $Q \neq \emptyset$, $f(w)\, [\twoheadrightarrow]\, Q$ holds if and only if $\exists\, t \in T_\Sigma, a \in A \;\; Q = f(wat)$, since:

$$\begin{aligned} f(w)\,[\twoheadrightarrow]\,Q \iff{}& \exists\, t \in T_\Sigma, a \in A \;\; Q = \{q' \in \mathcal{XS} : q \in f(w), q \twoheadrightarrow^a q', \mathrm{cterm}(q') = t\} \\ \iff{}& \exists\, t \in T_\Sigma, a \in A \;\; Q = \{q' \in \mathcal{XS} : t_0 \mathbin{@} \alpha = q_0 \twoheadrightarrow^{a_1} \cdots \twoheadrightarrow^{a_n} q_n \twoheadrightarrow^a q', \\ & \qquad \mathrm{cterm}(q') = t, \mathrm{cterm}(q_k) = w_k\} = f(wat) \end{aligned}$$

The relation $R = \{(t_0 w, f(t_0 w)) : w \in (T_\Sigma \cup A)^*, f(t_0 w) \neq \emptyset\}$ is the bisimulation we are looking for. Clearly, $(t_0, \{t_0 \mathbin{@} \alpha\}) \in R$ and $\ell_{\mathrm{last}}(ws) = \ell(s) = \ell(\mathrm{cterm}(Q))$ if $(ws, Q) \in R$. Given two words $v, w \in (T_\Sigma \cup A)^+$, R only relates them to $f(v)$ and $f(w)$, respectively. $(v, w) \in U$ implies $w = vat$ by definition of U, and then $f(v)\, [\twoheadrightarrow]\, f(w)$ follows from the previous paragraph. Given two non-empty sets Q and Q' such that $Q\, [\twoheadrightarrow]\, Q'$, and a word w with $f(w) = Q$, we must find a w' such that $(w, w') \in R$ and $f(w') = Q'$. However, we already have it thanks to the previous paragraph and $f(w)\, [\twoheadrightarrow]\, Q'$, since there is some a and t such that $f(wat) = Q'$. It remains to prove that $(w, wat) \in U$, i.e. $(a, t) \in \lambda(w)$, but since there are no failed states in $\mathcal{M}'_\alpha$, any step of the semantics must be allowed by the strategy. □

Lemma B.15. *For any partial context c, execution π, and $k \in \mathbb{N}$, $|\mathrm{Enter}(c, \pi)| - k \leq |\mathrm{Enter}(c, \pi^k)| \leq |\mathrm{Enter}(c, \pi)|$ and $|\mathrm{Leave}(c, \pi)| - k \leq |\mathrm{Leave}(c, \pi^k)| \leq |\mathrm{Leave}(c, \pi)|$, assuming $\infty - k = \infty$.*

Proof. These relations follow from the definition of Enter and Leave, because the membership of an index $n \in \mathrm{Enter}(c, \pi)$ only depends on π_n and π_{n+1}, i.e. it is equivalent to $0 \in \mathrm{Enter}(c, \pi_n \pi_{n+1})$. Since $\pi^k_n = \pi_{n+k}$, the function $f(n) = n + k$ is a bijection from $\mathrm{Enter}(c, \pi^k)$ to $\mathrm{Enter}(c, \pi) \cap [k, \infty)$, and so the cardinality of the first is bounded by the second without the intersection. Moreover, $\mathrm{Enter}(c, \pi)$ may include at most k indices more than the same set for π^k. The same happens for Leave. □

Lemma B.16. *Given two executions π and π' such that $\pi_k = \mathrm{subterm}(x : \pi'_{m_k}, \dots, t) @ c$ for all $k \in \mathbb{N}$ and an unbounded sequence $(m_k)_{k=0}^{\infty}$ with $0 \leq m_{k+1} - m_k \leq 1$, and given a context c', $\mathrm{Enter}(c'xc, \pi) = \mathrm{Enter}(c', \pi')$, and the same happens for* Leave.

Proof. The indices in $\mathrm{Enter}(c', \pi')$ are m_k for $k \in \mathrm{Enter}(c'xc, \pi)$ by the definition of these sets, because only the position $x\,\mathrm{pos}(c)$ is involved. The same happens for Leave. □

Proposition 4.8 (page 109). *Given an execution π of the semantics, π iterates finitely iff for every partial context $\alpha^* c \in \pi$, $\mathrm{Enter}(\alpha{*}\, c, \pi)$ is finite or $\mathrm{Leave}(\alpha{*}\, c, \pi)$ is not finite. Moreover, this happens iff $\mathrm{Enter}(\alpha{*}\, c, \pi)$ is finite or $\mathrm{Enter}(c', \pi)$ is infinite, where c' is the longest suffix of c whose top element is the context of a recursive call or ε if there is none.*

Proof. Take $\pi \in \mathrm{Ex}^{\omega}(\alpha, t)$, and let π' be the expansion of its intermediate $\rightarrow_{s,c}$ steps out of the $\twoheadrightarrow$ transitions. There must be a monotone sequence $(m_k)_{k=0}^{\infty}$ such that $\pi_k = \pi'_{m_k}$. We will first prove both implications of the first statement by a contrapositive argument, and then the second statement.

Suppose π' iterates forever for some iteration $\alpha\, {*c}$, we should prove that $\mathrm{Enter}(\alpha\, {*c}, \pi)$ is infinite and $\mathrm{Leave}(\alpha\, {*c}, \pi)$ is finite. Three cases are possible according to Definition 3.5. In the third one, there must be a k such that π^k iterates forever because of the other cases. The finiteness of Enter and Leave for π^k does not change by Lemma B.15, so we can assume without loss of generality that we are in one of the other cases. If $c = c_1 x c_2$ contains a variable, the execution necessarily happens in a subterm state and the definition falls in the second case. However, by Lemma B.16, the sets Enter and Leave for c_1 and the subexecution are as finite as those for the whole execution, so we can concentrate on the first case without loss of generality. Then, $\pi_k = t_k @ c_k \alpha\, {*c}$ for all $k \in \mathbb{N}$ and some t_k, c_k. Since $\alpha\, {*c}$ is maintained all along the execution, $\mathrm{Leave}(\alpha\, {*c}, \pi) = \emptyset$, so it is finite. Moreover, since there are infinitely many indices k such that $\pi'_k = t_k @ \alpha\, {*c}$ and $\pi'_{k+1} = t_k @ \alpha\alpha\, {*c}$, we can take infinitely many l such that $\pi_l = \pi'_{m_l} \rightarrow^*_c \pi'_k \twoheadrightarrow \pi_{l+1}$ for such k. These are clearly in Enter with $q = \pi'_k$ and $q' = \pi'_{k+1}$ by definition.

Suppose $\mathrm{Enter}(\alpha\, {*c}, \pi)$ is infinite and $\mathrm{Leave}(\alpha\, {*c}, \pi)$ is finite, we should prove π' iterates forever for $\alpha\, {*c}$. Take k_0 as the minimum element of $\mathrm{Enter}(\alpha\, {*c}, \pi)$ greater than $\max \mathrm{Leave}(\alpha\, {*c}, \pi)$. The execution π^{k_0+1} satisfies $\mathrm{Leave}(\alpha\, {*c}, \pi^{k_0+1}) = \emptyset$ and the iteration has been entered between k_0 and $k_0 + 1$, so the context of $\alpha\, {*c}$ is active all along $\pi'^{m_{k_0+1}}$. It is enough to prove that π^{k_0+1} iterates forever by the third item of its definition, so let π be that suffix. If the context $c = c_1 x c_2$ contains a variable, the iteration takes place in a subterm and we can proceed with the execution inside this position by Lemma B.16. Hence, we are necessarily in the first case with $\pi_k = t_k @ c_k \alpha\, {*c}$, because the context $\alpha\, {*c}$ is never left. Consider $n \in \mathrm{Enter}(\alpha\, {*c}, \pi)$, by definition, there must be a q such that $\pi_n = t_n @ c_n \alpha\, {*c} \rightarrow^c q$ and $\alpha\, {*c} \in q$, so q can only be $t_n @ \alpha\, {*c}$. Then there is a $q' = t_n @ c'$, of this form because $\mathrm{pos}(c) = \varepsilon$, such that $q \rightarrow_c q'$ and $c \sqsubsetneq_s c'$. It can only be $q' = t_n @ \alpha\alpha\, {*c}$. Hence, for each n there must be a k with $m_n \leq k \leq m_{n+1}$ such that $\pi'_k = t_n @ \alpha\, {*c}$ and $\pi'_{k+1} = t_n @ \alpha\alpha\, {*c}$, and this is precisely the condition to iterate forever.

The second statement changes the $\mathrm{Leave}(\alpha\, {*c}, \pi)$ by $\mathrm{Enter}(c', \pi)$ for a partial context $c' \sqsubseteq_s c$ that includes the environment of all recursive calls in c (this is slightly more general than the statement). For a strategy to be popped, as in the indices of Leave, infinitely often, it must have been replenished infinitely often too. The stack can only grow by the effect of iterations and strategy calls. In the first case, the context $\alpha\, {*c}$ would have been recovered infinitely often by another iteration $\beta*$ within c where β contains $\alpha*$. However, if $\beta*$ is not refilled in turn infinitely often, it is iterating forever. In the second case, if a recursive strategy replaces the iteration infinitely often, it must be called infinitely often and so the stack must grow from the point where it is called, so Enter must be infinite. □

Proposition 4.9 (page 110). *Given a strategy α and an initial term t, the Streett automaton produced from $\mathcal{M}_{\alpha,t}$ as indicated in Section 4.5 accepts the language $E_K(\alpha, t)$.*

Proof. The key is that Proposition 4.8 can be applied to the executions of that automaton. However, since conditions are imposed on states instead of transitions we may need to duplicate them to avoid ambiguities. This can probably be avoided by directing the application of control steps, but we do not care about this efficiency concern in this proof. Let $M = (Q^2, S, \delta, (t \mathbin{@} \alpha, t \mathbin{@} \alpha), F)$ be such an automaton with $Q \subseteq \mathcal{XS}$ the finite set of reachable execution states from $t \mathbin{@} \alpha$, $S = \mathrm{cterm}(Q) \subseteq T_\Sigma$ the finite subset of terms visited, and $\delta((r, q), t) = \{(q, q') \in \mathcal{XS} : q \twoheadrightarrow q'\}$ if $t = \mathrm{cterm}(q)$ and $\delta((r, q), t) = \emptyset$ otherwise. Its acceptance condition is $F = \{(E_k, L_k)\}_{k=1}^n$ where E_k and L_k are the states where each iteration $\alpha_k* \, c_k$ enters and leaves respectively. More precisely, $(q, q') \in E_k$ if $0 \in \mathrm{Enter}(\alpha_k* \, c_k, qq')$. It is clear from the definition that the word accepted by a run ρ in M is $\mathrm{cterm}(\pi_2(\rho))$ where π_2 is the projection on the second coordinate.

By definition of $\mathrm{Ex}^\omega(\alpha, t)$, the second projections of the runs in the automaton are the executions in this set. We should prove that the projections of the accepting runs are its subset $\mathrm{Ex}^\omega_K(\alpha, t)$ of executions that iterate finitely. Since $E_K(\alpha, t) = \mathrm{cterm}(\mathrm{Ex}^*(\alpha, t)) \cup \mathrm{cterm}(\mathrm{Ex}^\omega_K(\alpha, t))$, it is enough to prove what we want assuming that finite executions are handled separately as usual. Given an arbitrary run ρ in M, we know it has the form $\rho_0 = (\pi_0, \pi_0)$ and $\rho_k = (\pi_{k-1}, \pi_k)$ for $k \geq 1$ and an execution π in $\mathcal{XS}$. Proposition 4.8 says that $\mathrm{Enter}(\alpha_k* \, c_k, \pi)$ is finite or $\mathrm{Leave}(\alpha_k* \, c_k, \pi)$ is infinite for all k iff π iterates finitely. In M, ρ is accepting if $\{m : \rho_m \in E_k\}$ is finite or $\{m : \rho_m \in L_k\}$ is infinite for all k. For all $n \in \mathbb{N}$, $n \in \mathrm{Enter}(\alpha_k* \, c_k, \pi)$ (or Leave) iff $\rho_{n+1} \in E_k$ (or L_k). In effect, it is a general property that $n \in \mathrm{Enter}(c, \pi)$ iff $0 \in \mathrm{Enter}(c, \pi_n \pi_{n+1})$, and $\rho_k \in E_k$ iff $\mathrm{Enter}(\alpha_k* \, c_k, \pi_{k-1}\pi_k)$. Hence, the cardinality of Enter and Leave coincides respectively with the cardinality of the set of positions of ρ in E_k and L_k for all k. This means that ρ is accepting iff $\pi = \pi_2(\rho)$ iterates finitely, iff $\mathrm{cterm}(\pi_2(\rho))$ belongs to $E_K(\alpha, t)$. □

Proposition 4.10 (page 110). *Given a Maude strategy expression α, an LTL property φ, and assuming that the iterations that occur in the reachable states from $t \mathbin{@} \alpha$ are numbered from 1 to n, $(\mathcal{M}, E_K(\alpha, t)) \vDash \varphi$ is equivalent to $\mathcal{M}^K_{\alpha,t} \vDash \psi \to \varphi$ where $\mathcal{M}^K_{\alpha,t} := (\mathcal{XS} \times \mathcal{P}(AP'), \twoheadrightarrow', \{t \mathbin{@} \alpha\}, AP \cup AP', \ell')$ with $AP' = \{e_1, l_1, \ldots, e_n, l_n\}$, $\ell'((q, U)) = \ell(q) \cup U$, $(q, U) \twoheadrightarrow' (q', U')$ if $q \twoheadrightarrow_{\mathrm{Sol}} q'$, and U' contains e_k or l_k if the iteration k has been entered or left from q to q' (as explained above), $\psi = \bigwedge_{k=1}^n \psi_k$, and $\psi_k = \Diamond\Box\neg e_k \,\vee\, \Box\Diamond l_k$.*

Proof. Observe that $\mathcal{M}^K_{\alpha,t}$ only extends $\mathcal{M}_{\alpha,t}$ with additional atomic propositions that are appended to each state. The states in the first coordinate may appear repeated with a different second coordinate, but this is convenient for the case where there are iterations in multiple substates of a `matchrew`. For example, the execution

$$\begin{aligned}\mathrm{subterm}(x : t_x \mathbin{@} \alpha*, y : t_y \mathbin{@} \texttt{rep}; t, c) \mathbin{@} s &\to_c \mathrm{subterm}(x : t_x \mathbin{@} \alpha\, \alpha*, y : t_y \mathbin{@} \texttt{rep}; t, c) \mathbin{@} s \\ &\to_s \mathrm{subterm}(x : t_x \mathbin{@} s'\alpha*, y : t_y \mathbin{@} \texttt{rep}; t, c) \mathbin{@} s,\end{aligned}$$

where `rep` is defined as $\beta*$`;rep`, may be completed to an infinite execution that executes $\beta*$ infinitely often but iterating finitely each, visiting the last state above infinitely often. This path must be accepted as it iterates finitely, but the last state is labeled with e_1 (for $\alpha*$) and it appears infinitely often, so it will not satisfy ψ. Recall that $E_K(\alpha)$ is the subset of $E(\alpha)$ where executions iterate finitely. Concentrating on an iteration k, the second clause of $\psi_k = \Diamond\Box\neg e_k \vee \Box\Diamond l_k$ means that a path leaves the iteration infinitely often according to the LTL semantics, and the first claims that the path enters the iteration k finitely often. Hence, ψ is the Streett condition in Proposition 4.8. Now, writing ℓ for $L_\Pi \circ \text{cterm}$,

$$\begin{aligned}
(\mathcal{M}, E_K(\alpha, t)) \vDash \varphi &\iff \forall w \in E_K(\alpha, t) \;\; \mathcal{M}, L_\Pi(w) \vDash \varphi \\
&\iff \forall \pi \in \Gamma_{\mathcal{M}_{\alpha,t}} \;\; \pi \text{ iterates finitely implies } \mathcal{M}_{\alpha,t}, \ell(\pi) \vDash \varphi \\
&\iff \forall (\pi, \rho) \in \Gamma_{\mathcal{M}^K_{\alpha,t}} \;\; \mathcal{M}^K_{\alpha,t}, (\ell(\pi_k) \cup \rho_k)_k \vDash \psi \text{ implies } \mathcal{M}^K_{\alpha,t}, \ell(\pi) \vDash \varphi \\
&\iff \mathcal{M}^K_{\alpha,t} \vDash \psi \to \varphi
\end{aligned}$$

where $\mathcal{M}^K_{\alpha,t}, (\ell(\pi) \cup \rho_k)_k \vDash \varphi$, $\mathcal{M}^K_{\alpha,t}, \ell(\pi) \vDash \varphi$, and $\mathcal{M}_{\alpha,t}, \ell(\pi) \vDash \varphi$ are equivalent since they only refer to atoms in AP. □

Proposition 4.11 (page 113). *Given a Kripke structure $\mathcal{K}$ and an LTL formula φ, $E = \{\pi \in \Gamma_\mathcal{K} : \mathcal{K}, \ell(\pi) \vDash \varphi\}$ is an ω-regular strategy and its calculation is* PSPACE*-complete in the size of $\mathcal{K}$ and the Büchi automaton of φ.*

Proof. Let $A = (Q, \mathcal{P}(AP), \delta, q_0, F)$ be the Büchi automaton for φ and $B = (S \times \{\iota\}, S, \zeta, \iota, S \cup \{\iota\})$ be the system automaton for $\mathcal{K}$ with $\zeta(\iota, s) = \{s : s \in I\}$ and $\zeta(s, s') = \{s' : s \to s'\}$. Since their alphabets are different, we define the product automaton as $G = (Q \times (S \cup \{\iota\}), S, \xi, (q_0, \iota), F \times (S \cup \{\iota\}))$ with $\xi((q, x), s) = \{(q', x') : q' \in \delta(q, \ell(s)), x' \in \zeta(x, s)\}$. This recognizes all executions of $\mathcal{K}$ whose propositional traces are allowed by φ, which are exactly those included in E by definition.

In effect, if w is accepted by G and by definition of ξ, the first projection gives an accepting run r in A for $\ell(w)$ and the second projection gives an accepting run r' in B for w. Hence, $w \in \Gamma_\mathcal{K}$ and $\mathcal{K}, \ell(w) \vDash \varphi$, so $w \in E$. In the opposite direction, $w \in E$ implies that ιw is accepted in B and $\ell(w)$ in A, so combining their two runs we get an accepting run in G for w by the definition of ξ. Finally, $E = L(G)$ is ω-regular and since calculating it involves calculating an arbitrary intersection, it is a PSPACE-complete problem. □

Proposition 5.1 (page 124). *Given a labeled (or unlabeled when appropriate) transition system $(S, A, \to)$ and the additional data required by the methods,* `uniform`*,* `uaction`*,* `metadata`*, and* `term` *produce a discrete-time Markov chain (S, P, P_0). On the other hand, the methods* `mdp-uniform` *and* `mdp-metadata` *produce a Markov decision process (S, A, P, P_0).*

Proof. We should only check that the probabilistic transition function P satisfies $\sum_{s \to s'} P(s, s') = 1$ for every $s \in S$ or $\sum_{s \to^a s'} P(s, a, s')$ for every $s \in S$ and $a \in A$. This is clear by definition in `uniform` and `mdp-uniform`, and also in `metadata`, `mdp-metadata`, and `term`. In the latter case, we should assume that the term is always successfully reduced to a numeric literal, so that the weight function w is well defined. For the case of `uaction`, let us sum the expression $P(s, s')$ over all $s' \in S$. Only the successors of s contribute to the sum, since $P(s, s') = 0$ otherwise. In the first addend, for a fixed $a \in A_p$, $p(a)$ appears once for each $s \to^a s'$, but this is $\text{sc}(s, a)$ times. Hence, the first part is simplified to $\sum_{a \in A_p} p(a) = \text{ap}(s)$. The same happens with the weights of the second addend, which yield $\sum_{a \in A_w} w(a) = \text{tw}(a)$. Finally, $\sum_{s' \in S} P(s, s') = \text{ap}(s) + 1 - \text{ap}(s) = 1$. □

Bibliography

[AAD+06] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Comput.*, 18(4):235–253, 2006.

[AAE06] D. Angluin, J. Aspnes, and D. Eisenstat. Stably computable predicates are semilinear. In E. Ruppert and D. Malkhi, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, pages 292–299. ACM, 2006.

[ABL+19] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, and R. Zunino. Developing secure bitcoin contracts with BitML. In M. Dumas, D. Pfahl, S. Apel, and A. Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 1124–1128. ACM, 2019.

[Abr94] S. Abramsky. Domain theory. In *Handbook of Logic in Computer Science*. S. Abramsky, D. M. Gabbay, and T. S. E. Malbaum, editors. Volume 3. Clarendon Press, 1994, pages 1–168.

[AC09] O. Agrigoroaiei and G. Ciobanu. Rewriting logic specification of membrane systems with promoters and inhibitors. In [Roş09], pages 5–22.

[ACD90] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 414–425. IEEE Computer Society, 1990.

[ACL05] O. Andrei, G. Ciobanu, and D. Lucanu. Executable specifications of P systems. In G. Mauri, G. Păun, M. J. Pérez-Jiménez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing, 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*, volume 3365 of *Lecture Notes in Computer Science*, pages 126–145. Springer, 2005.

[ACL07] O. Andrei, G. Ciobanu, and D. Lucanu. A rewriting logic framework for operational semantics of membrane systems. *Theor. Comput. Sci.*, 373(3):163–181, 2007.

[AHK02] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[AL09] O. Andrei and D. Lucanu. Strategy-based proof calculus for membrane systems. In [Roş09], pages 23–43.

[ALY20] M. Avanzini, U. D. Lago, and A. Yamada. On probabilistic term rewriting. *Sci. Comput. Program.*, 185, 2020.

[AMP+21] L. Aguirre, N. Martí-Oliet, M. Palomino, and I. Pita. Strategies in Conditional Narrowing Modulo SMT Plus Axioms. Technical report 02/21, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2021.

[AMS06] G. A. Agha, J. Meseguer, and K. Sen. PMaude: rewrite-based specification language for probabilistic object systems. In A. Cerone and H. Wiklicky, editors, *Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages, QAPL 2005, Edinburgh, UK, April 2-3, 2005*, volume 153(2) of *Electronic Notes in Theoretical Computer Science*, pages 213–239. Elsevier, 2006.

[AP18] G. Agha and K. Palmskog. A survey of statistical model checking. *ACM Trans. Model. Comput. Simul.*, 28(1):6:1–6:39, 2018.

[ASB$^+$94] A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Equivalences for fair Kripke structures. In S. Abiteboul and E. Shamir, editors, *Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings*, volume 820 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 1994.

[Bar14] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 131 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 2nd edition, 2014. ISBN: 978-0-444-87508-2.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1): 217–248, 1992.

[BBH$^+$13] J. Barnat, L. Brim, V. Havel, J. Havlícek, J. Kriho, M. Lenco, P. Rockai, V. Still, and J. Weiser. DiVinE 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In [SV13b], pages 863–868.

[BBK$^+$07] E. Balland, P. Brauner, R. Kopetz, P.-. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In F. Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007.

[BCD$^+$09] T. Bourdier, H. Cirstea, D. J. Dougherty, and H. Kirchner. Extensional and intensional strategies. In M. Fernández, editor, *Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009, Brasilia, Brazil, 28th June 2009*, volume 15 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–19, 2009.

[BD94] L. Bachmair and N. Dershowitz. Equational inference, canonical proofs, and proof orderings. *J. ACM*, 41(2):236–276, 1994.

[BEJ$^+$17] M. Blondin, J. Esparza, S. Jaax, and P. J. Meyer. Towards efficient verification of population protocols. In E. M. Schiller and A. A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 423–430. ACM, 2017.

[BEJ$^+$18] M. Blondin, J. Esparza, S. Jaax, and A. Kucera. Black ninjas in the dark: formal analysis of population protocols. In A. Dawar and E. Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 1–10. ACM, 2018.

[BEM13] K. Bae, S. Escobar, and J. Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In F. van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[BG05] O. Bournez and F. Garnier. Proving positive almost-sure termination. In J. Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2005.

[BGK+19] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, volume 11428 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2019.

[BHK+20] P. Berenbrink, D. Hammer, D. Kaaser, U. Meyer, M. Penschuck, and H. Tran. Simulating population protocols in sub-constant time per interaction. In F. Grandoni, G. Herman, and P. Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, 16:1–16:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[BJ14] N. Bulling and W. Jamroga. Comparing variants of strategic ability: how uncertainty and memory influence general properties of games. *Auton. Agents Multi Agent Syst.*, 28(3):474–518, 2014.

[BJM97] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer, 1997.

[BK02] O. Bournez and C. Kirchner. Probabilistic rewrite strategies. applications to ELAN. In S. Tison, editor, *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2378 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2002.

[BK08] C. Baier and J.-. Katoen. *Principles of Model Checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[BKK+01] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *Int. J. Found. Comput. Sci.*, 12(1):69–95, 2001.

[BKV+08] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2): 52–70, 2008.

[BM15] K. Bae and J. Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. *Sci. Comput. Program.*, 99:193–234, 2015.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. ISBN: 978-1-139-17275-2.

[BÖ13] L. Bentea and P. C. Ölveczky. A probabilistic strategy language for probabilistic rewrite theories and its application to cloud computing. In N. Martí-Oliet and M. Palomino, editors, *Recent Trends in Algebraic Development Techniques, 21st International Workshop, WADT 2012, Salamanca, Spain, June 7-10, 2012, Revised Selected Papers*, volume 7841 of *Lecture Notes in Computer Science*, pages 77–94. Springer, 2013.

[Bri19] R. Brijder. Computing with chemical reaction networks: a tutorial. *Nat. Comput.*, 18(1):119–137, 2019.

[BT15] C. Baier and C. Tinelli, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, 2015. Springer.

[BV07] C. Braga and A. Verdejo. Modular structural operational semantics with strategies. In R. van Glabbeek and P. D. Mosses, editors, *Proceedings of the Third Workshop on Structural Operational Semantics, SOS 2006, Bonn, Germany, August 26, 2006*, volume 175(1) of *Electronic Notes in Theoretical Computer Science*, pages 3–17. Elsevier, 2007.

[BW18] J. C. Bradfield and I. Walukiewicz. The mu-calculus and model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 871–919. Springer, 2018.

[Cas20] A. Casagrande. pyModelChecking. A simple Python model checking package, 2020.

[CBC15] C. Chareton, J. Brunel, and D. Chemouil. A logic with revocable and refinable strategies. *Inf. Comput.*, 242:157–182, 2015.

[CCG+02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: an opensource tool for symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[CCK20] M. Ceska, C. Chau, and J. Kretínský. SeQuaiA: A scalable tool for semi-quantitative analysis of chemical reaction networks. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 653–666. Springer, 2020.

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. ISBN: 978-3-540-71940-3.

[CDE+20] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott. *Maude Manual v3.1*. October 2020.

[CDF+11] J. Clément, C. Delporte-Gallet, H. Fauconnier, and M. Sighireanu. Guidelines for the verification of population protocols. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 215–224. IEEE Computer Society, 2011.

[CDL+08] F. Calzolai, R. De Nicola, M. Loreti, and F. Tiezzi. TAPAs: A tool for the analysis of process algebras. *Trans. Petri Nets Other Model. Concurr.*, 1:54–70, 2008.

[CDS16] R. Cummings, D. Doty, and D. Soloveichik. Probability 1 computation with chemical reaction networks. *Nat. Comput.*, 15(2):245–261, 2016.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CEL$^+$96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In [Mes96], pages 65–89.

[CFP$^+$18] F. Corradini, F. Fornari, A. Polini, B. Re, and F. Tiezzi. A formal approach to modeling and verification of business process collaborations. *Sci. Comput. Program.*, 166:35–70, 2018.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999. ISBN: 978-0-262-03883-6.

[CHP10] K. Chatterjee, T. A. Henzinger, and N. Piterman. Strategy logic. *Inf. Comput.*, 208(6): 677–693, 2010.

[CHV$^+$18] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10575-8.

[CLM$^+$14] P. Cermák, A. Lomuscio, F. Mogavero, and A. Murano. MCMAS-SLK: A model checker for the verification of strategy logic specifications. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 525–532. Springer, 2014.

[CLM15] P. Cermák, A. Lomuscio, and A. Murano. Verifying and synthesising multi-agent systems against one-goal strategy logic specifications. In B. Bonet and S. Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 2038–2044. AAAI Press, 2015.

[CM17] G. Caltais and B. Meyer. On the verification of SCOOP programs. *Sci. Comput. Program.*, 133:194–215, 2017.

[CM96] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In [Mes96], pages 126–148.

[CM97] M. Clavel and J. Meseguer. Internal strategies in a reflective logic. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction*, pages 1–12, Townsville, Australia, 1997.

[Coq21] The Coq Development Team. The Coq proof assistant, 2021.

[CPP06] G. Ciobanu, M. J. Pérez-Jiménez, and G. Păun, editors. *Applications of Membrane Computing*. Natural Computing Series. Springer, 2006. ISBN: 978-3-540-25017-3.

[CPR06] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *J. Univers. Comput. Sci.*, 12(11):1618–1650, 2006.

[Cra12] S. Cranen. Model checking the FlexRay startup phase. In M. Stoelinga and R. Pinger, editors, *Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*, volume 7437 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2012.

[CS93] K. Crowley and R. S. Siegler. Flexible strategy use in young children's tic-tac-toe. *Cogn. Sci.*, 17(4):531–561, 1993.

[CV07] E. Csuhaj-Varjú and G. Vaszil. P systems with string objects and with communication by request. In G. Eleftherakis, P. Kefalas, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers*, volume 4860 of *Lecture Notes in Computer Science*, pages 228–239. Springer, 2007.

[CY95] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4):857–907, 1995.

[Dav01] M. Davis. *Engines of Logic. Mathematicians and the Origin of the Computer*. 2001. ISBN: 978-0-393-32229-3.

[DEE+20] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott. Programming and symbolic computation in Maude. *J. Log. Algebraic Methods Program.*, 110, 2020. 58 pages.

[DEL04] F. Durán, S. Escobar, and S. Lucas. New evaluation commands for Maude within Full Maude. In [Mar04], pages 263–284.

[DF02] R. Diaconescu and K. Futatsugi. Logical foundations of CafeOBJ. *Theor. Comput. Sci.*, 285(2):289–318, 2002.

[Dij82] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*. Texts and Monographs in Computer Science. Springer, 1982, pages 60–66.

[DJL+15] A. David, P. G. Jensen, K. G. Larsen, M. Mikucionis, and J. H. Taankvist. UPPAAL STRATEGO. In [BT15], pages 206–211.

[DKN+06] M. Duflot, M. Z. Kwiatkowska, G. Norman, and D. Parker. A formal analysis of bluetooth device discovery. *Int. J. Softw. Tools Technol. Transf.*, 8(6):621–632, 2006.

[DLF+16] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 - A framework for LTL and ω-automata manipulation. In C. Artho, A. Legay, and D. Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129, 2016.

[DM09] Y. Deng and J.-F. Monin. Verifying self-stabilizing population protocols with Coq. In W.-N. Chin and S. Qin, editors, *TASE 2009, Third IEEE International Symposium on Theoretical Aspects of Software Engineering, 29-31 July 2009, Tianjin, China*, pages 201–208. IEEE Computer Society, 2009.

[DRÁ11] F. Durán, C. Rocha, and J. M. Álvarez. Towards a Maude Formal Environment. In G. Agha, O. Danvy, and J. Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science*, pages 329–351. Springer, 2011.

[DV90] R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer, 1990.

[EGL+17] J. Esparza, P. Ganty, J. Leroux, and R. Majumdar. Verification of population protocols. *Acta Informatica*, 54(2):191–215, 2017.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[EHJ+20] J. Esparza, M. Helfrich, S. Jaax, and P. J. Meyer. Peregrine 2.0: explaining correctness of population protocols through stage graphs. In D. V. Hung and O. Sokolsky, editors, *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, volume 12302 of *Lecture Notes in Computer Science*, pages 550–556. Springer, 2020.

[ELL04] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Int. J. Softw. Tools Technol. Transf.*, 5(2-3):247–267, 2004.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985. ISBN: 3-540-13718-1.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier and MIT Press, 1990.

[EMM+07] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In M. Archer, T. B. de la Tour, and C. Muñoz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, August 16, 2006*, volume 174(11) of *Electronic Notes in Theoretical Computer Science*, pages 3–25. Elsevier, 2007.

[EMM+20] S. Eker, N. Martí-Oliet, J. Meseguer, R. Rubio, and A. Verdejo. The Maude strategy language. *Manuscript*, 2020.

[EMM+21] S. Eker, N. Martí-Oliet, J. Meseguer, I. Pita, R. Rubio, and A. Verdejo. Strategy language for Maude. 2021. URL: `http://maude.ucm.es/strategies`.

[EMM09] S. Escobar, C. A. Meadows, and J. Meseguer. Maude-NPA: cryptographic protocol analysis modulo equational properties. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2009.

[EMS04] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proceedings of the Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19-21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 162–187. Elsevier, 2004.

[FCM+04] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In R. Alur and D. A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

[Fin14a] O. Finkel. Ambiguity of omega-languages of Turing machines. *Log. Methods Comput. Sci.*, 10(3), 2014.

[Fin14b] O. Finkel. Topological complexity of context-free ω-languages: A survey. In N. Dershowitz and E. Nissan, editors, *Language, Culture, Computation. Computing - Theory and Technology - Essays Dedicated to Yaacov Choueka on the Occasion of His 75th Birthday, Part I*, volume 8001 of *Lecture Notes in Computer Science*, pages 50–77. Springer, 2014.

[FKP19] M. Fernández, H. Kirchner, and B. Pinaud. Strategic port graph rewriting: an interactive modelling framework. *Math. Struct. Comput. Sci.*, 29(5):615–662, 2019.

[FL79] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[Flo67] R. W. Floyd. Assigning meaning to programs. In J. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–31. American Mathematical Society, 1967.

[GI14] M. Gheorghe and F. Ipate. A kernel P systems survey. In A. Alhazov, S. Cojocaru, M. Gheorghe, Y. Rogozhin, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing - 14th International Conference, CMC 2013, Chișinău, Republic of Moldova, August 20-23, 2013, Revised Selected Papers*, volume 8340 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2014.

[GKI+15] M. Gheorghe, S. Konur, F. Ipate, L. Mierla, M. E. Bakir, and M. Stannett. An integrated model checking toolset for kernel P systems. In G. Rozenberg, A. Salomaa, J. M. Sempere, and C. Zandron, editors, *Membrane Computing - 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers*, volume 9504 of *Lecture Notes in Computer Science*, pages 153–170. Springer, 2015.

[GMM20] G. D. Giacomo, B. Maubert, and A. Murano. Nondeterministic strategies and their refinement in strategy logic. In D. Calvanese, E. Erdem, and M. Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, pages 294–303, 2020.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[Gog84] J. A. Goguen. Parameterized programming. *IEEE Trans. Software Eng.*, 10(5):528–544, 1984.

[GV04] A. Groce and W. Visser. Heuristics for model checking Java programs. *Int. J. Softw. Tools Technol. Transf.*, 6(4):260–276, 2004.

[GV13] G. D. Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In F. Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 854–860. IJCAI/AAAI, 2013.

[GWM+00] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*. J. Goguen and G. Malcolm, editors. Springer, 2000, pages 3–167.

[HBS73] C. Hewitt, P. B. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In N. J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973.

[Her20] A. Hernández Cerezo. *Strategies for implementing SAT algorithms in rewriting logic*. Spanish. Bachelor's thesis, Universidad Complutense de Madrid, 2020.

[HJ94] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects Comput.*, 6(5):512–535, 1994.

[HJK+21] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk. The probabilistic model checker STORM. *Int. J. Softw. Tools Technol. Transf.*, 23(4):1–22, 2021.

[HKK+13] Y.-. Hwong, J. J. A. Keiren, V. J. J. Kusters, S. J. J. Leemans, and T. A. C. Willemse. Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. *Sci. Comput. Program.*, 78(12):2435–2452, 2013.

[HM80] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 1980.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN: 0-13-153271-5.

[Hol+21] G. Holzmann et al. Spin - Formal verification. 2021. URL: `https://spinroot.com`.

[HPY97] G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 23–31. DIMACS/AMS, 1997.

[HR86] H. J. Hoogeboom and G. Rozenberg. Infinitary languages: basic theory an applications to concurrent systems. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency, Overviews and Tutorials*. Volume 224, Lecture Notes in Computer Science, pages 266–342. Springer, 1986.

[Hue81] G. P. Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *J. Comput. Syst. Sci.*, 23(1):11–21, 1981.

[HVO07] M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Using Maude and its strategies for defining a framework for analyzing Eden semantics. In S. Antoy, editor, *Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2006, Seattle, WA, USA, August 11, 2006*, volume 174(10) of *Electronic Notes in Theoretical Computer Science*, pages 119–137. Elsevier, 2007.

[IS04] M. Ionescu and D. Sburlan. On P systems with promoters/inhibitors. *J. UCS*, 10(5): 581–599, 2004.

[JM14] W. Jamroga and A. Murano. On module checking and strategies. In A. L. C. Bazzan, M. N. Huhns, A. Lomuscio, and P. Scerri, editors, *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5-9, 2014*, pages 701–708. IFAAMAS/ACM, 2014.

[Jur00] M. Jurdzinski. Small progress measures for solving parity games. In H. Reichel and S. Tison, editors, *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, Proceedings*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2000.

[Kal21] O. Kallenberg. *Foundations of Modern Probability*. Probability Theory and Stochastic Modelling. Springer, 2nd edition, 2021. ISBN: 978-3-030-61871-1.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra. Proceedings of a Conference Held at Oxford Under the Auspices of the Science Research Council Atlas Computer Laboratory, 29th August to 2nd September 1967*, pages 263–297. Pergamon Press, 1970.

[KKK08] C. Kirchner, F. Kirchner, and H. Kirchner. Strategic computation and deduction. In C. Benzmüller, C. E. Brown, J. Siekmann, and R. Statman, editors, *Reasoning in Simple Type Theory. Festchrift in Honour of Peter B. Andrews on His 70th Birthday*. Volume 17, Studies in Logic and the Foundations of Mathematics, pages 339–364. College Publications, 2008.

[KLM$^+$15] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: high-performance language-independent model checking. In [BT15], pages 692–707.

[KM95] H. Kirchner and P.-. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995, Proceedings*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer, 1995.

[KNP11] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[KNS02] M. Z. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In H. Hermanns and R. Segala, editors, *Process Algebra and Probabilistic Methods, Performance Modeling and Verification, Second Joint International Workshop PAPM-PROBMIV 2002, Copenhagen, Denmark, July 25-26, 2002, Proceedings*, volume 2399 of *Lecture Notes in Computer Science*, pages 169–187. Springer, 2002.

[Kow79] R. A. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, 1979.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[KVW00] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000.

[Lam80] L. Lamport. "Sometime" is sometimes "not never" - on the temporal logic of programs. In P. W. Abrahams, R. J. Lipton, and S. R. Bourne, editors, *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, pages 174–185. ACM Press, 1980.

[LÁÖ15] D. Lepri, E. Ábrahám, and P. C. Ölveczky. Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. *Sci. Comput. Program.*, 99:128–192, 2015.

[Lec96] U. Lechner. Object-oriented specifications of distributed systems in the μ-calculus and Maude. In [Mes96], pages 385–404.

[Les90] P. Lescanne. Implementations of completion by transition rules + control: ORME. In H. Kirchner and W. Wechler, editors, *Algebraic and Logic Programming, Second International Conference, Nancy, France, October 1-3, 1990, Proceedings*, volume 463 of *Lecture Notes in Computer Science*, pages 262–269. Springer, 1990.

[Lok10] A. J. Lokta. Contribution to the theory of periodic reactions. *J. Phys. Chem.*, 14(3): 271–274, 1910.

[LQR17] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: an open-source model checker for the verification of multi-agent systems. *Int. J. Softw. Tools Technol. Transf.*, 19(1): 9–30, 2017.

[LT16] C. Löding and A. Tollkötter. Transformation between regular expressions and ω-automata. In P. Faliszewski, A. Muscholl, and R. Niedermeier, editors, *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*, volume 58 of *LIPIcs*, 88:1–88:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[Luc20] S. Lucas. Context-sensitive rewriting. *ACM Comp. Surv.*, 53(4), 2020.

[Luc21] S. Lucas. Applications and extensions of context-sensitive rewriting. *J. Log. Algebraic Methods Program.*, 121, 2021.

[Luc92] É. Lucas. *Recréations mathématiques*. Albert Blanchard, 2nd edition, 1992. ISBN: 2-85367-121-6.

[LYP$^+$] K. G. Larsen, W. Yi, P. Petterson, A. David, B. Nielsen, A. Skou, J. Håkansson, J. I. Rasmussen, P. Krcál, U. Larsen, M. Mikucionis, L. Mokrushin, et al. UPPAAL. URL: `http://www.uppaal.org/`.

[LZ74] B. H. Liskov and S. N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[Mal98] G. Malkin. RIP Version 2. RFC 2453, Internet Engineering Task Force, November 1998. 39 pages.

[Mar04] N. Martí-Oliet, editor. *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, 2004. Elsevier.

[McM93] K. L. McMillan. *Symbolic model checking*. Kluwer, 1993. ISBN: 978-0-7923-9380-1.

[Men42] L. F. Menabrea. Sketch of the analytical engine invented by Charles Babbage. Translated by A. A. Byron. *Bibliothèque Universelle de Genève*, (82), 1842.

[Mes07] J. Meseguer. The Temporal Logic of Rewriting. Technical report UIUCDCS-R-2007-2815, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[Mes08] J. Meseguer. The temporal logic of rewriting: A gentle introduction. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 354–382. Springer, 2008.

[Mes12] J. Meseguer. Twenty years of rewriting logic. *J. Log. Algebr. Program.*, 81(7-8):721–781, 2012.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[Mes96] J. Meseguer, editor. *Proceedings of the First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3-6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, 1996. Elsevier.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. ISBN: 3-540-10235-3.

[MJ84] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *IEEE Ann. Hist. Comput.*, 6(2):139–143, 1984.

[MK06] M. Marin and T. Kutsia. Foundations of the rule-based system ρlog. *J. Appl. Non Class. Logics*, 16(1-2):151–168, 2006.

[MM02] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*. Volume 9, pages 1–87. Kluwer Academic Publishers, 2nd edition, 2002.

[MM96] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In [Mes96], pages 190–225.

[MMP+14] F. Mogavero, A. Murano, G. Perelli, and M. Y. Vardi. Reasoning about strategies: on the model-checking problem. *ACM Trans. Comput. Log.*, 15(4):34:1–34:47, 2014.

[MMR+21] E. Martin-Martin, M. Montenegro, A. Riesco, J. Rodríguez-Hortalá, and R. Rubio. Verification of ROS Navigation using Maude. In N. Martí-Oliet, editor, *Actas de las XX Jornadas de Programación y Lenguajes (PROLE 2021)*. Sistedes, 2021. 10 pages.

[MMV04] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In [Mar04], pages 417–441.

[MMV09] N. Martí-Oliet, J. Meseguer, and A. Verdejo. A rewriting semantics for Maude strategies. In [Roş09], pages 227–247.

[MPV07] N. Martí-Oliet, M. Palomino, and A. Verdejo. Strategies and simulations in a semantic framework. *J. Algorithms*, 62(3-4):95–116, 2007.

[MPV12] N. Martí-Oliet, M. Palomino, and A. Verdejo. Rewriting logic bibliography by topic: 1990-2011. *J. Log. Algebr. Methods Program.*, 81(7-8):782–815, 2012.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[MSC+13] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In [SV13b], pages 696–701.

[MT04] I. A. Mason and C. L. Talcott. IOP: the InterOperability Platform & IMaude: an interactive extension of Maude. In [Mar04], pages 315–333.

[MVM20] Ó. Martín, A. Verdejo, and N. Martí-Oliet. Compositional specification in rewriting logic. *Theory Pract. Log. Program.*, 20(1):44–98, 2020.

[NASA11] NASA Engineering and Safety Center. National Highway Traffic Safety Administration Toyota Unintended Acceleration Investigation. Technical report TI-10-00618, 2011.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN: 3-540-43376-7.

[Oga17] K. Ogata. Model checking the iKP electronic payment protocols. *J. Inf. Secur. Appl.*, 36:101–111, 2017.

[Ölv14] P. C. Ölveczky. Real-Time Maude and its applications. In S. Escobar, editor, *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, volume 8663 of *Lecture Notes in Computer Science*, pages 42–79. Springer, 2014.

[Pad87] P. Padawitz. Strategy-controlled reduction and narrowing. In P. Lescanne, editor, *Rewriting Techniques and Applications, 2nd International Conference, RTA-87, Bordeaux, France, May 25-27, 1987, Proceedings*, volume 256 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 1987.

[Pău02] G. Păun. *Membrane Computing: An Introduction*. Natural computing series. Springer, 2002. ISBN: 978-3-540-43601-0.

[Pea84] J. Pearl. *Heuristics. Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984. ISBN: 978-0-201-05594-8.

[Pel93] D. A. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Rheinisch-Westfaelisches Institut für Instrumentelle Mathematik and der Universitaet Bonn, 1962.

[Plo04] G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Methods Program.*, 60-61:3–15, 2004.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487, 1976.

[PM02] I. Pita and N. Martí-Oliet. A Maude specification of an object-oriented model for telecommunication networks. *Theor. Comput. Sci.*, 285(2):407–439, 2002.

[PMV05] M. Palomino, N. Martí-Oliet, and A. Verdejo. Playing with Maude. In S. Abdennadher and C. Ringeissen, editors, *Proceedings of the 5th International Workshop on Rule-Based Programming, RULE 2004, Aachen, Germany, June 1, 2004*, volume 124 of number 1 in *Electronic Notes in Theoretical Computer Science*, pages 3–23. Elsevier, 2005.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[PP04] D. Perrin and J.-E. Pin. *Infinite words. Automata, semigroups, logic and games*, volume 141 of *Pure and applied mathematics series*. Elsevier Morgan Kaufmann, 2004. ISBN: 978-0-12-532111-2.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 179–190. ACM Press, 1989.

[QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[Rie18] A. Riesco. Model checking parameterized by the semantics in Maude. In J. P. Gallagher and M. Sulzmann, editors, *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings*, volume 10818 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2018.

[RMP+19a] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Model checking strategy-controlled rewriting systems. In H. Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, 34:1–34:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[RMP+19b] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Model checking strategy-controlled rewriting systems (extended version). Technical report 02/19, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2019.

[RMP+19c] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Parameterized strategies specification in Maude. In J. Fiadeiro and I. Țuțu, editors, *Recent Trends in Algebraic Development Techniques. 24th IFIP WG 1.3 International Workshop, WADT 2018, Egham, UK, July 2–5, 2018, Revised Selected Papers*, volume 11563 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2019.

[RMP+20a] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Metalevel transformation of strategies. In *7th International Workshop on Rewriting Techniques for Program Transformation and Evaluation, WPTE 2020, Paris, France*, 2020. 10 pages.

[RMP+20b] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Simulating and model checking membrane systems using strategies in Maude. In *7th International Workshop on Rewriting Techniques for Program Transformation and Evaluation, WPTE 2020, Paris, France*, 2020. 10 pages.

[RMP+20c] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Strategies, model checking and branching-time properties in Maude. In S. Escobar and N. Martí-Oliet, editors, *Rewriting Logic and Its Applications - 13th International Workshop, WRLA 2020, Virtual Event, October 20-22, 2020, Revised Selected Papers*, volume 12328 of *Lecture Notes in Computer Science*, pages 156–175. Springer, 2020.

[RMP+21a] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Model checking strategy-controlled systems in rewriting logic. *Automat. Softw. Eng.*, 2021. Accepted. 55 pages.

[RMP+21b] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Strategies, model checking and branching-time properties in Maude. *J. Log. Algebr. Methods Program.*, 123, 2021. 28 pages.

[RMP+21c] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. The semantics of the Maude strategy language. Technical report 01/21, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2021.

[RMP+22a] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Metalevel transformation of strategies. *J. Log. Algebr. Methods Program.*, 124, 2022. 36 pages.

[RMP+22b] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Simulating and model checking membrane systems using strategies in Maude. *J. Log. Algebr. Methods Program.*, 124, 2022. 42 pages.

[Roşu09] G. Roşu, editor. *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008*, volume 238(3) of *Electronic Notes in Theoretical Computer Science*, 2009. Elsevier.

[RS10] G. Rosu and T.-F. Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Methods Program.*, 79(6):397–434, 2010.

[RSV06] F. Rosa-Velardo, C. Segura, and A. Verdejo. Typed mobile ambients in Maude. In H. Cirstea and N. Martí-Oliet, editors, *Proceedings of the 6th International Workshop on Rule-Based Programming, RULE 2005, Nara, Japan, April 23, 2005*, volume 147(1) of *Electronic Notes in Theoretical Computer Science*, pages 135–161. Elsevier, 2006.

[Rub21a] R. Rubio. Experimental language bindings for Maude. FADoSS. 2021. URL: `https://fadoss.github.io/maude-bindings`.

[Rub21b] R. Rubio. Unified Maude model-checking tool (umaudemc). FaDoSS. 2021. URL: `https://github.com/fadoss/umaudemc`.

[RW84] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. In A. Bensoussan and J. Lions, editors, *Proceedings of the Sixth International Conference on Analysis and Optimization of Systems Nice, June 19–22, 1984*, volume 63 of *Lecture Notes in Control and Information Sciences*, pages 475–498. Springer, 1984.

[RW90] D. Ratner and M. K. Warmuth. The (n^2-1)-puzzle and related relocation problems. *J. Symb. Comput.*, 10(2):111–138, 1990.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.

[SP07] G. Santos-García and M. Palomino. Solving Sudoku puzzles with rewriting rules. In G. Denker and C. Talcott, editors, *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006*, volume 176 of number 4 in *Electronic Notes in Theoretical Computer Science*, pages 79–93. Elsevier, 2007.

[SPV09] G. Santos-García, M. Palomino, and A. Verdejo. Rewriting logic using strategies for neural networks: an implementation in Maude. In J. M. Corchado, S. Rodríguez, J. Llinas, and J. M. Molina, editors, *International Symposium on Distributed Computing and Artificial Intelligence, DCAI 2008, University of Salamanca, Spain, 22th-24th October 2008*, volume 50 of *Advances in Soft Computing*, pages 424–433. Springer, 2009.

[SS71] D. Scott and C. Strachey. Towards a mathematical semantics for programmign languages. Technical report PRG-6, Oxford University Computing Laboratory, 1971.

[Sta97] L. Staiger. ω-languages. In *Handbook of Formal Languages, Volume 3: Beyond Words*. G. Rozenberg and A. Salomaa, editors. Springer, 1997, pages 339–387.

[SV13a] S. Sebastio and A. Vandin. MultiVeStA: statistical model checking for discrete event simulators. In A. Horváth, P. Buchholz, V. Cortellessa, L. Muscariello, and M. S. Squillante, editors, *7th International Conference on Performance Evaluation Methodologies and Tools, ValueTools '13, Torino, Italy, December 10-12, 2013*, pages 310–315. ICST/ACM, 2013.

[SV13b] N. Sharygina and H. Veith, editors. *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, 2013. Springer.

[SWI20] The SWIG Developers. Simplified wrapper and interface generator. 2020. URL: `http://www.swig.org/`.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2): 146–160, 1972.

[TB18] A. S. Tanenbaum and H. Bos. *Modern operating systems*. Pearson, 4th edition, 2018. ISBN: 978-1-292-06142-9.

[TCP08] D. Thomas, S. Chakraborty, and P. K. Pandya. Efficient guided symbolic reachability using reachability expressions. *Int. J. Softw. Tools Technol. Transf.*, 10(2):113–129, 2008.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. ISBN: 978-0-521-39115-3.

[Tho89] W. Thomas. Computation tree logic and regular ω-languages. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *Lecture Notes in Computer Science*, pages 690–713. Springer, 1989.

[vBen10] J. van Benthem. *Modal Logic for Open Minds*. CSLI Lecture Notes. Stanford University, 2010. ISBN: 978-1-575-86658-1.

[VM11] A. Verdejo and N. Martí-Oliet. Basic completion strategies as another application of the Maude strategy language. In S. Escobar, editor, *Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 17–36, 2011.

[VS85] M. Y. Vardi and L. J. Stockmeyer. Improved upper and lower bounds for modal logics of programs: preliminary report. In R. Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 240–251. ACM, 1985.

[Wan04] B.-Y. Wang. μ-calculus model checking in Maude. In [Mar04], pages 135–152.

[Wig08] J. E. Wiggelinkhuizen. *Feasibility of formal model checking in the Vitatron environment*. Master's thesis, Technische Universiteit Eindhoven, 2008.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction*. Foundations of Computing. The MIT Press, 1993. ISBN: 978-0-262-23169-5.

[WKV14] G. Wachsmuth, G. D. P. Konat, and E. Visser. Language design with the Spoofax Language Workbench. *IEEE Softw.*, 31(5):35–43, 2014.

[Zie98] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.