UNIVERSITÀ DEGLI STUDI DI ROMA
"TOR VERGATA"

Facoltà di Ingegneria
Corso di Laurea in Informatica

Tesi di Laurea

Algorithms and Tools for the Formal Verification of Concurrent Systems

Relatore
Professor **Alberto Pettorossi**

Candidato
Lorenzo Clemente

Anno Accademico 2006-2007

Prefazione

Sistemi concorrenti con un elevato grado di complessità sono comunemente presenti nello sviluppo di circuiti digitali e nei protocolli di comunicazione. Il maggior ostacolo alla realizzazione di sistemi concorrenti di grandi dimensioni può essere individuato nello scarso livello di confidenza che viene percepito rispetto al loro comportamento. Tecniche di verifica comunemente utilizzate come la *simulazione* ed il *testing* possono analizzare solo *alcuni* dei possibili comportamenti del sistema, lasciando aperta la possibiltà di errori nei percorsi non esplorati. Una via alternativa viene fornita dalla *verifica formale*, dove tutti i possibili comportamenti sono esplorati in maniera *esaustiva*.

Tra i diversi approcci alla verifica formale che sono stati proposti, quello noto come *model-checking* è particolarmente interessante. In questo approccio la specifica di correttezza viene scritta come formula in una qualche *logica temporale* e viene verificato se il sistema sia un modello per tale formula. Due vantaggi degni di particolare rilievo che l'approccio del model-checking ha rispetto ad altri metodi formali (come, ad esempio, il theorem-proving) sono: il model-checking è *completamente automatico* e, qualora un bug nel modello venisse individuato, viene generato un *controesempio*, ovvero un comportamento del sistema che mostra come esso violi la proprietà in esame. Inoltre, con l'introduzione di tecniche *simboliche* per la rappresentazione di sistemi di grandi dimensioni, l'applicabilità del model-checking si è notevolmente estesa, permettendone l'uscita dal mondo prettamente accademico per entrare nella pratica industriale (specialmente per lo sviluppo di *hardware*).

Lo scopo di questa tesi è duplice. In primo luogo, viene presentata la nuova logica temporale ω-**CTL**, che consente di esprimere vincoli di *fairness* direttamente nel linguaggio di specifica. La logica ω-**CTL** è un'estensione della logica **CTL**, ottenuta interpretando i quantificatori di percorso **E** e **A** relativamente ad i percorsi descritti da *espressioni ω-regolari*. Per esempio, se a è un'espressione ω-regolare e f una (generica) formula, allora $\mathbf{EG}(a, f)$ indica che esiste un percorso descritto da a dove f vale in ogni stato di tale percorso. In questo modo è possibile esprimere vincoli di *fairness* direttamente nella specifica, permettendo l'utilizzo di vincoli diversi in formule diverse (ed anche all'interno di parti diverse della stessa formula). Questo approccio a grana fine per la *fairness* si rivela utile in quanto proprietà diverse necessitano, in generale, di vincoli diversi. Inoltre, è possibile ottenere dei vantaggi in termini di prestazioni nel momento in cui una formula viene provata sotto vincoli più deboli e computazionalmente meno onerosi. Per effettuare il model-checking di specifiche scritte in ω-**CTL**, viene fornita una traduzione in μ-calculus, per poi utilizzare un algoritmo di model-checking specifico per il μ-calculus.

In secondo luogo, è stato ideato un nuovo algoritmo di model-checking per il μ-calculus, implementato nel model-checker simbolico NuSMV [CCG$^+$02]. Tale algoritmo si basa sugli algoritmi di Emerson e Lei [EL86] e Browne et al. [BCJ$^+$94]. In particolare, l'algoritmo di Browne et al. ha complessità temporale $O(n^{d/2})$ (dove n è la dimensione del modello e d è una misura della dimensione della specifica[1]) e risulta essere l'algoritmo con la minore complessità asintotica attualmente noto. Se si confronta la complessità $O(n^{d/2})$ di Browne et al. con la complessità $O(n^d)$ di Emerson e Lei, si nota che l'algoritmo di [BCJ$^+$94] ha una complessità temporale che è circa la radice quadrata rispetto a [EL86]. Sfortunatamente, l'algoritmo di Browne et al. necessita di memorizzare un numero esponenziale di risultati intermedî, mentre l'algoritmo di Emerson e Lei richiede solo spazio lineare. Dunque, l'algoritmo di Browne et al. viene generalmente ritenuto di scarso interesse pratico. In questo lavoro è stato fatto un tentativo per migliorare questa situazione, barattando una minor efficienza temporale per un minor consumo di memoria. La complessità temporale asintotica dell'algoritmo che viene proposto è maggiore di quella di Browne et al., ma non supera quella dell'algoritmo di Emerson e Lei. Inoltre, la complessità spaziale è quadratica e, visto che vengono utilizzati più valori intermedî rispetto a Emerson e Lei, ci si aspetta che questo nuovo algoritmo si comporti meglio del loro nelle applicazioni pratiche.

[1]Infatti, d è nota come *alternation depth* della formula in μ-calculuse conta il numero di alternanze negli operatori di punto fisso.

Preface

Complex concurrent systems arise naturally in the development of digital circuits and communication protocols. It is commonly agreed that one of the main obstacle to the construction of large concurrent systems lies in our limited ability to have a sufficiently high degree of confidence of the correctness of their behaviour. Classical verification techniques like *simulation* and *testing* only explore *some* of the possible behaviours, leaving the possibility of uncovered bugs into the remaining traces. An attractive alternative to simulation and testing is the approach of *formal verification*, where all behaviours are *exhaustively* explored.

Among several proposed approaches to formal verification, the approach of *model-checking* is quite interesting. Here a specification is written in some *temporal logic* formula and the system is checked to see whether or not it is a model of the formula. Two remarkable advantages of model-checking over other approaches to formal verification (like theorem-proving) are: model-checking is *fully automatic* and, when a bug is uncovered, a *counterexample* behaviour is given, which explains how the system can evolve and eventually reach an erroneous state. Moreover, with the advent of *symbolic* techniques, huge systems can be represented by implicit enumeration and verified with iterative methods.

The contribution of this thesis is twofold. On the one hand, we present the novel temporal logic ω-**CTL**, which expresses fairness constraints within the specification language. ω-**CTL** is an extension of **CTL**, where path quantifiers **E** and **A** are interpreted over paths described by ω*-regular expressions*. For example, for a regular expression a and a formula f, $\mathbf{EG}(a, f)$ is true when there exists an *a-path* in which f continuously holds. In this way we can express fairness constraints directly into temporal logic formulae, enabling the use of different fairness assumptions into different formulae (and even in different part of the *same* formula). Such finer-grained use of fairness is useful because not all properties need the same set of fairness constraints. Moreover, we can gain in efficiency when proving a formula under simpler fairness assumptions. For the purpose of model-checking, we give a translation of ω-**CTL** into μ-calculus, and use a μ-calculus model-checking algorithm.

On the other hand, we designed a new μ-calculus model-checking algorithm and we implemented it in the NuSMV symbolic model-checker of [CCG$^+$02]. Our algorithm is based on the algorithms by Emerson and Lei's [EL86] and Browne's et al. [BCJ$^+$94]. In particular, Browne's et al. presented a μ-calculus algorithm of time complexity $O(n^{d/2})$ (where n is the size of the model and d is a measure of the size of the specification[2]), which is the algorithm with the best known asymptotic running time. Comparing $O(n^{d/2})$ with the complexity $O(n^d)$ of Emerson and Lei's algorithm, we see that the algorithm of [BCJ+94] requires about the square root of the time of [EL86]. Unfortunately, Browne's algorithm requires exponential space for storing intermediate results (while Emerson and Lei's requires linear space), hence it is generally argued that Browne's algorithm is of little practical use. We tried to improve on this, trading space for efficiency. The asymptotic running time of our algorithm is not as good as Browne's, but is as least as good as Emerson and Lei's. Moreover, we use quadratic space and, since we use more intermediate values than Emerson and Lei's, we expect that our algorithm performs better than theirs in practice.

Thesis structure

The remainder of this thesis is organized as follows. In Chapter 1 we introduce the problem of formally verifying concurrent systems, along with examples in NuSMV. In Chapter 2 we survey the most common temporal logics used for specifying correctness properties and in Chapter 3 we present basic symbolic model-checking techniques. Our new temporal logic is developed in Chapter 4 and our model-checking algorithm is presented in Chapter 5. Implementation issues and details are then given in Chapter 6. In Chapter 7 we present experimental results and, finally, in Chapter 8 we draw some conclusions and discuss some directions for future research.

[2]In fact, d is the *alternation depth* of the μ-calculus formula, i.e. the number of alternating fixpoints.

Contents

Part I

Background: Temporal Logic and Model-Checking

Chapter 1

Introduction

In this chapter we present the problem of formal verification. In Section 1.1 we show some motivating examples and in Section 1.2 we introduce the model-checking approach to formal verification. In Section 1.3 we briefly survey other approaches to formal verification. In Section 1.4 we introduce the notion of *fairness* and in Section 1.5 we develop the basics of the fixpoint theory, which we will need throughout the thesis.

1.1 Motivation

Practical methods for formally verifying the correctness of hardware and software systems are needed in applications where failure is not acceptable. Examples are electronic commerce, telephone networks, air traffic control systems, health-care instruments, space devices, and many others, where failure can result in huge financial losses or, worst of all, human-life loss. These *critical systems* require to be highly reliable and formal verification methods have been proven fruitful in uncovering subtle bugs.

The Ariane Rocket

An (in)famous example where formal methods would probably had been useful is represented by the failure of the Ariane Rocket. The launch of the *Ariane 5 Flight 501* was planned on 4 June 1996. The rocket was self-destructing only 37 seconds after the launch, because of a malfunction in the control software. This was arguably one of the most expensive computer bugs in history. The following is taken from [Gle96].

The ESA (European Space Agency) set up a committee and the causes of failure were investigated. They discovered that pieces of code inherited from its predecessor, Ariane 4, were used without proper validation. The explosion was ultimately caused by a data conversion from 64-bit floating point to 16-bit signed integer value, which caused a processor trap (operand error) in the guidance system, which immediately shut down. The same happened to the backup unit, which was running identical software, on identical hardware. The 64-bit value that caused overflow represented the sideways velocity of the rocket and engineers had decided that no overflow was possible. Hence, they turned off exception handling mechanisms for performance purposes. Unfortunately, Ariane 5 was faster than Ariane 4 and overflow, indeed, occurred. The bad data propagated into the on-board computer, which believed that a high attitude deviation had occurred. The on-board computer tried to make a large correction acting on the main engine nozzle and on the booster nozzles, but this caused the launcher to disintegrate. Moreover, the calculation that eventually led to explosion, was actually useless once the rocket was in the air. In fact, it only served to align the system before the launch, so it should have been turned off! But engineers decided to leave this function running for the first 40 seconds of flight, in order to make it easy to restart the system in the event of a brief hold in the count-down. This led to the loss of 10 years of work and $7 billions.

Another (in)famous example worth of being mentioned is the *Pentium FDIV bug*, which caused Intel to replace all flawed processors, at the cost of $500 millions. It can be argued that also in this case the use of formal methods could had been extremely cost-saving.

1.2 The model-checking approach

Model-checking is a practical approach to the formal verification of finite state concurrent systems. It is based on the exhaustive search for faulty behaviours. When systems are finite-state, i.e., containing a finite number of states, the method is fully algorithmic and complete. The properties we are interested in are expressed in *temporal logic* specifications (see Chapter 2). When a property is false, model-checking algorithms can explain the user why this is the case, by showing a faulty behaviour of the system, i.e., a trace that falsifies the property. Counterexamples have proven to be extremely useful in developing higher confidence in the system.

One of the main limitations of the applicability of model-checking is the *state-explosion* problem. When a system is composed by many concurrent parts that can take independent actions, the set of possible states can reach astronomically large values. (In general, n parallel processes can result into approximately 2^n states.) Hence, techniques aimed at handling such huge sets of states are needed. On such technique is the *symbolic approach*, where states are not represented individually. Indeed, the so-called ROBDDs (Reduced Ordered Binary Decision Diagrams) data structures are used to *implicitly* represent subsets of the state space [Bry86, McM92b]. This allowed researchers to go further, and larger and larger systems where investigated, up to 10^{20} [BCM$^+$92] and 10^{120} [BCL91] states.

Some examples of the successful application of model-checking to real-world problems are:

- *IEEE Futurebus+ Standard*. The IEEE Futurebus+ cache coherence standard was verified in 1992 at CMU (Carnegie Mellon University) by one of the leading research group in model-checking. Clarke and his colleagues found for the first time several uncatched bugs in an approved IEEE standard, and they showed the applicability of model-checking in examples of practical interest. They obtained analogous results with the *IEEE SCI* (Scalable Coherent Interface).

- *PowerScale.* IBM PowerScale is a multiprocessor architecture based on PowerPC. It consists of a bus arbiter, a memory controller and a set of processors. In 1995, the research group at the Verimag Laboratory formalized the arbiter protocol and they proved the correctness of the architecture.
- *Concurrent protocols* A lot of communication protocols have been verified with model-checking, including the *Alternating Bit Protocol* and many mutual exclusion algorithms (Dekker, Peterson, Knuth, ...). We will return to concrete examples of similar protocols in Chapter 7.

1.3 Other approaches to formal verification

Model-checking is not the only way in formal methods. Other relevant approaches are the (related) automata-theoretic approach, theorem-proving and transformation methodologies, as explained below.

Automata-theoretic model-checking

The *automata-theoretic* approach to model-checking applies to finite state systems and (in first approximation) on linear-time properties [Var07a]. The property is usually, but not necessarily, written as a **LTL** formula ϕ, or in a similar linear-time formalism. Then, the formula ϕ is converted into a finite state automaton on infinite words $A_{\neg\phi}$ (called the *complementary automaton*), that accepts precisely the computations that falsify ϕ. The type of automaton considered is often the one defined by Büchi in the early 60's [Büc62]. Hence, the system M is composed with the automaton $A_{\neg\phi}$ to obtain the *cross product* $A_{M,\neg\phi}$, with the property that ϕ is violated iff there exists an accepting run in $A_{M,\neg\phi}$. So, the general problem of model-checking has been reduced to testing the emptiness of the language generated by the automaton $A_{M,\neg\phi}$, for which several efficient graph-based, explicit algorithms exist. In the simplest form, those algorithms are based on testing *fair reachability* on graphs, in the following way:

- The graph (or automaton) is decomposed into its *maximal strongly connected components* (MSCCs), which can be done with a depth-first search in linear time wrt the size of the graph.

- It is checked if a reachable accepting state belongs to a MSCC, i.e., if there is a cycle on an accepting state. If this is the case, then the automaton is non-empty.

If the automaton is non-empty, then an accepting word in the form of a lasso is returned. A *lasso* is a finite prefix of states followed by a (repeating) finite cycle. Hence, the lasso provides a counter-example to the original formula ϕ, like in model-theoretic model-checking.

Theorem-proving

The *theorem-proving* technique involves formalizing the system and the properties of interest in some kind of mathematical logic. This logic is part of a formal system, composed of axioms and rules of inference. A property is verified by finding a derivation from axioms and applying rules of inference. Informally, a *derivation* (or *proof*) is a sequence of formulae, where each formula is either an axiom, or has been obtained by applying a rule of inference from previous formulae. If a property appears somewhere in a derivation, then that property is said to be a *theorem* in the formal system and verification can stop. *Theorem-provers* are programs that assist the user in constructing derivations. Theorem-provers are *interactive*, in the sense that they need user-guidance in constructing the proofs, and *mechanical*, in the sense that they are capable of automatically applying heuristic algorithms and reduction techniques during the verification process. The theorem-proving technique can also handle infinite-state systems. However, the problem of verifying infinite state systems is undecidable in general. Some notable examples of the success of theorem-proving are (taken from [CWA$^+$96]):

- The *SRT division algorithm*, where the correctness of the division algorithm were established with symbolic algebraic manipulation. An analogous technique could have been applied to the Pentium processor, avoiding the aforementioned, infamous FDIV bug.
- In general, *processor design*, where whole parts of "real-life" processors were formally verified. In particular, the following processors were formally verified with theorem-proving: PowerPC, System/390, Motorola 68020, AMD5K86, Motorola CAP (*Complex Arithmetic Processor*) and the Collinc Commercial Avionics AAMP5.

It is worth mentioning that an emerging research line involves combining the generality of theorem-proving with the automation of model-checking. In particular, model-checking would become one of the decision procedures that the theorem-prover applies when proving a property. This tight integration enables the use of powerful techniques, like abstraction (the theorem-prover could prove that the abstraction is sound), inductive reasoning and compositional verification. For further discussion, see [Sha97].

Program transformation

Another promising approach to formal verification is that of *program transformation*. In this setting, the system under investigation is specified as a *constraint logic program* P, while a given temporal property is specified by means of an atomic formula A. Unfortunately,e.g., it is not possible to directly check whether A belongs to the least Herbrand model of P with the usual SLDNF resolution, since it will loop forever for many temporal formulae of interest. Tabled resolution could be used instead, since it avoids looping by maintaining a table of previous predicate calls. However, neither tabled resolution nor SLDNF are suitable for *infinite-state* systems. Hence, in [FPP05] is presented an approach based on program transformation, as we explain. Properties are given in the **CTL** temporal logic. Instead of directly checking if A holds in P, P is transformed into a new program T with the repeated application of some *transformation rules*. The property P is identified with a new predicate $new0$. Moreover, the new program T has the property that A holds iff $new0$ belongs to the least Herbrand Model of T iff $new0$ occurs in T. Hence, to check if A holds in P, it suffices to see if $new0$ is in T. This method is sound for **CTL** properties and also complete for finite-state systems. Moreover, it is also capable of proving properties of several infinite-state systems which are used in practice.

1.4 Fairness constraints

We review here some of the most common fairness constraints used in practice (following [Eme95]). Fairness is often associated with fair scheduling, where we put restrictions on how the next process to execute is chosen. In general, we want to rule out unfair computation paths, for example those paths in which there are processes that are never executed.

Suppose there are n concurrently executed processes. With atomic propositions en_i, ex_i we mean that process i is enabled to run, i.e., can be executed, has been selected for execution, respectively. When we say that en_i occurs *almost everywhere* we mean that there exists a later time in which en_i holds forever. When we say that en_i occurs *infinitely often* we mean that it is not the case that $\neg en_i$ occurs almost everywhere, i.e., we must have an infinite number of occurrence of en_i. Common fair scheduling constraints are[1]:

- *Impartiality*: every process is executed infinitely often, e.g., fair computation paths contain each of ex_i infinitely often.
- *Weak fairness* or *Justice*: if a process is enabled almost everywhere, then it is executed infinitely often.
- *Strong fairness* or *Compassion*: if a process is enabled infinitely often, then it is executed infinitely often.

[1]For expressing fairness in temporal logic see 2.3, on page 19.

However, fairness constraints can also be useful outside scheduling. For example, in a mutual exclusion protocol we want to allow a process to remain in its critical section for a finite, though bounded, amount of time. Common modeling formalisms cannot directly express the property that the process must eventually exit its critical section, hence we need to put this restriction in the specification, with a fairness constraint ruling out computations in which a process remains in its critical section forever. This is an *impartiality* constraint, because we want the process to be outside the critical section infinitely often. Moreover, if we replace en_i with ask_i and ex_i with got_i, then we can also model the general situation in which a process asks for a resource and obtains it.

1.5 Complete lattices and fixpoints

In this section we review the theory of complete lattices and fixpoints of monotonic functions. We will state and prove the Knaster-Tarski theorem, as appeared in [Tar55].

A *lattice* is a pair $\langle A, \sqsubseteq \rangle$, where A is a nonempty set and $\sqsubseteq \subseteq A \times A$ is a partial order on A, i.e., a binary relation which is reflexive, anti-symmetric and transitive. Given any two elements $a, b \in A$, there exist:

- The least upper bound $a \sqcup b$ (lub), which satisfies:
  - upper bound of a, b: $a \sqsubseteq a \sqcup b$ and $b \sqsubseteq a \sqcup b$, and
  - least of upper bounds: for all c, $a \sqsubseteq c$ and $b \sqsubseteq c$ implies $a \sqcup b \sqsubseteq c$.
- The greatest lower bound $a \sqcap b$ (glb), which satisfies:
  - lower bound of a, b: $a \sqcap b \sqsubseteq a$ and $a \sqcap b \sqsubseteq b$, and
  - greatest of lower bounds: for all c, $c \sqsubseteq a$ and $c \sqsubseteq b$ implies $c \sqsubseteq a \sqcap b$.

Note that lubs and glbs are unique. In fact, if there were, say, two lubs x, x', each being a lub implies $x \sqsubseteq x'$ and $x' \sqsubseteq x$, hence $x = x'$.

We can extend the previous definition of lub and glb to any subset $B \subseteq A$. For finite subset, lubs and glbs exist being A a lattice. In this case, they can be defined by induction as:

- (Basis) $\bigsqcup \{a\} = a$,
- (Step) $\bigsqcup \{a_1, \ldots, a_{n+1}\} = \bigsqcup \{a_1, \ldots, a_n\} \sqcup a_{n+1}$.

The definition for glbs is dual. The existence of such lubs (glbs) can be easily proved by an induction on (the elements in) B.

For infinite subsets $B \subseteq A$, we need an additional property. A lattice is *complete* if it has lubs and glbs of *arbitrary* subsets of A. In particular, there are two special elements:

$$\top \doteq \bigsqcup A \quad \text{and} \quad \bot \doteq \bigsqcap A \ .$$

Given two elements $a \sqsubseteq b$, they define an *interval* $[a,b]$ with endpoints a and b:

$$[a,b] \doteq \{ \ x \ \mid a \sqsubseteq x \sqsubseteq b \ \} \ .$$

Clearly, $\langle [a,b], \sqsubseteq \rangle$ is a lattice. Also, it is complete if A is complete[2].

A function $f : A \mapsto A$ is *monotonic* if it respects $\sqsubseteq$, i.e., for all a, b, $a \sqsubseteq b$ implies $f(a) \sqsubseteq f(b)$. An element $a \in A$ is a *prefixpoint* of f if $f(a) \sqsubseteq a$ (we also say that f is *decreasing* on a), and a *postfixpoint* if $a \sqsubseteq f(a)$ (f is *increasing* on a). Finally, a is a *fixpoint* of f if it is both a prefixpoint and a postfixpoint, i.e., $f(a) = a$. Clearly, $\bot$ ($\top$) is a postfixpoint (resp. prefixpoint) of *every* function.

We are now ready for the following

Theorem 1.1 (Tarski, 1939)**.** *Let $\langle A, \sqsubseteq \rangle$ be a complete lattice and f a monotonic function on A. Let $Post \doteq \{ \ a \ \mid a \sqsubseteq f(a) \ \}$ be the set of postfixpoints of f, $Pre \doteq \{ \ a \ \mid f(a) \sqsubseteq a \ \}$ the set of prefixpoint and let $P \doteq Post \cap Pre$ be the set of fixpoints of f. Then P is nonempty and $\langle P, \sqsubseteq \rangle$ is a complete lattice. In particular, there are elements $\bigsqcup P, \bigsqcap P \in P$ such that:*

$$\bigsqcup P = \bigsqcup Post \quad \textit{and} \quad \bigsqcap P = \bigsqcap Pre \ .$$

Proof. Let

$$v \doteq \bigsqcap Pre = \bigsqcap \{ \ a \ \mid f(a) \sqsubseteq a \ \} \ . \tag{1.1}$$

For any $x \in Pre$, $v \sqsubseteq x$ and, by monotonicity, $f(v) \sqsubseteq f(x) \sqsubseteq x$. Hence $f(v) \sqsubseteq x$. A $f(v)$ is a lower bound of Pre,

$$f(v) \sqsubseteq v \ , \tag{1.2}$$

making v a prefixpoint of f, i.e., $v \in Pre$. Then, again by monotonicity, $f(f(v)) \sqsubseteq f(v)$, that is also $f(v)$ is a prefixpoint. As v is a lower bound on prefixpoints,

$$v \sqsubseteq f(v) \ . \tag{1.3}$$

Hence v is a fixpoint and, being the least prefixpoint, it is also the least fixpoint[3]:

$$v = \bigsqcap P = \bigsqcap Pre \ . \tag{1.4}$$

By considering the dual lattice $\langle A, \sqsupseteq \rangle$[4], with analogous considerations we can obtain the symmetric relation

$$\bigsqcup P = \bigsqcup Post \ . \tag{1.5}$$

Now, let Q be an arbitrary subset of P. We will prove that Q has a lub and a glb in the lattice $\langle P, \sqsubseteq \rangle$, that is we will prove that there exists a minimal

[2]In fact, the converse *does not* hold. $[a,b]$ can be complete also if A is not complete.

[3]Something analogous is true in general. For $X_1 \subseteq X_2 \subseteq A$, $\bigsqcap X_2 \sqsubseteq \bigsqcap X_1$ and $\bigsqcup X_1 \sqsubseteq \bigsqcup X_2$.

[4]Where $\sqsupseteq \ \doteq \ (\sqsubseteq)^{-1}$.

(maximal) fixpoint of f which is an upper bound (lower bound, resp.) on Q. Consider the complete lattice

$$\langle\ [\ \bigsqcup Q,\ \top\ ],\ \sqsubseteq\ \rangle\ . \tag{1.6}$$

(Note that, in general, $\bigsqcup Q$ is not a fixpoint.) Then, for any $x \in Q$, $x \sqsubseteq \bigsqcup Q$ and applying f we get $f(x) \sqsubseteq f(\bigsqcup Q)$. But x is a fixpoint, so $x \sqsubseteq f(\bigsqcup Q)$ and hence

$$\bigsqcup Q \sqsubseteq f(\bigsqcup Q)\ , \tag{1.7}$$

being $f(\bigsqcup Q)$ and upper bound on Q. So, for any $x' \ \sqsupseteq\ \bigsqcup Q$, $\bigsqcup Q \sqsubseteq f(x')$. That is, if we restrict f to $[\ \bigsqcup Q,\ \top\ ]$, we obtain a new monotonic function $f' : [\ \bigsqcup Q,\ \top\ ] \mapsto [\ \bigsqcup Q,\ \top\ ]$. Hence, we can apply equation 1.4 to f', which says that f' has a least fixpoint q s.t. $\bigsqcup Q \sqsubseteq q$. Clearly, q is also a fixpoint of f, and in fact the least fixpoint of f that is an upper bound on Q. We have proved that Q has a lub in $\langle P, \sqsubseteq\rangle$. By an analogous argument for the glb, we conclude that $\langle P, \sqsubseteq\rangle$ is a complete lattice.

$\square$

Chapter 2

Temporal logic specifications

In this chapter we introduce *temporal logic*, as a formal means of reasoning about concurrent systems. In Section 2.1 we present a brief history of temporal logic, starting from A. Church and ending with A. Pnueli and the inventors of model-checking. In Section 2.1.1 we present the debate between the branching and the linear approaches to the underlying nature of time. In Section 2.2 we introduce the temporal logic **CTL***, and in Section 2.3 we introduce its fragments **CTL** and **LTL**. In Section 2.4 we present a very expressive branching time logic, the μ-calculus. Moreover, we give a sufficient condition ensuring that predicate transformers defined via μ-calculus formulae commute with arbitrary unions and intersections (theorems 2.2 and 2.3). Finally, in Section 2.5 we show how **CTL** formulae can be translated into μ-calculus. Moreover, we give pointers to the (more involved) translation of the whole **CTL***.

2.1 Brief history of Temporal Logic[1]

Classical logic Hardware verification started at the end of 50's, when A. Church used logic to specify sequential circuits (in [Chu57]). He proposed classical logic interpreted over infinite word structures as a logic of time. In his paper, Church posed two problems:

- *Decision problem*: it is the first formulation of what is now commonly called the *Model-checking* problem: given a circuit and a sentence, does the sentence hold in every trace of the circuit? Alternatively, is the circuit a model of the sentence?
- *Synthesis problem*: given a sentence, construct a circuit which is a model of the sentence.

[0]Talking about his husband A. Prior, after his death.
[1]The following was inspired from [Var07b].

While each one of these two problems is the reciprocal of the other, at the beginning the latter received much more attention than the former (and it is object of ongoing research). In fact, from a logical point of view, the decision problem is equivalent to the validity problem. We can construct a characteristic formula ϕ_C for a given circuit C s.t. ϕ_C encodes all the behaviours of C. Then we can reduce the decision problem of (another formula) ψ as the validity of $\phi_C \Rightarrow \psi$. So the decision problem is not so interesting from a logical perspective, and remained quiescent until 80's.

Temporal logic The birth of modern temporal logic is unanimously credited to A. Prior, which called it *tense logic*[2]. Prior was a philosopher and he was interested in formalizing tense logic; he published his ideas in "Time and Modality" [Pri57], the milestone book on tense logic.

It is interesting to note that the branching vs linear time dichotomy born at the very beginning of temporal logic. While Prior assumed a *linear* structure of time, shortly thereafter S. Kripke proposed a *branching* interpretation of future; we will further illustrate this ongoing debate later on, see 2.1.1, on page 13.

Although Prior himself made some early observations on the benefits of applying temporal logic on digital circuits, the revolutionary step was made by A. Pnueli in 1977 [Pnu77], which eventually resulted in the winning of the Turing Award in 1996, "for seminal work introducing temporal logic into computing science and for outstanding contributions to program and systems verification". In this paper, Pnueli proposed future linear temporal logic as a means of reasoning about non-terminating programs. His logic is now commonly called **LTL**.

Model-checking The decision problem raised by Church in 50's gained a new life in 80's, when E. M. Clarke and E. A. Emerson [CE81], and, independently, J.-P. Queille and J. Sifakis [QS81], introduced algorithms for solving the (finite state) *model-checking* problem for branching-time logics[3]. In particular, Emerson and Clarke introduced the branching time logic **CTL**—which was inspired by the logic UB of Ben-Ari, Manna and Pnueli[4]—and they also gave a model-checking algorithm of linear complexity. Moreover, in the same paper, they considered also the synthesis problem for concurrent programs: given a set of temporal logic formulae defining the behaviour of the system, they exploited the small-model property for **CTL** formulae[5] in order to construct a concurrent system of small size satisfying the given specification.

[2]The name *temporal logic* is due to A. Urquart and N. Rescher in [UR71].

[3]The "discovery" of model-checking eventually resulted in the winning of the 2007 A. M. Turing Award (widely considered the most prestigious award in computing) by Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis, "for their original and continuing research in a quality assurance process known as Model Checking. Their innovations transformed this approach from a theoretical technique to a highly effective verification technology that enables computer hardware and software engineers to find errors efficiently in complex system designs". See `http://www.acm.org/news/featured/turing-2007`.

[4]The logic UB was the first logic to consider the use of explicit path quantifiers **E** and **A**.

[5]The *small-model property* means that if f is satisfiable then there exists a *finite* model for f, of size bounded by some function of (the size of) f.

After [CE81], a lot of other algorithms for model checking were proposed. In particular, in [CES86] the original model-checking algorithm was extended to include *fairness constraints*, while in [BCM$^+$92] was initiated the *symbolic* approach, where the state space is represented in an implicit manner through the use of the ROBDD data structure (invented in [Bry86]).

On the other hand, also model-checking for linear-time logic evolved after the original work of Pnueli. In [LP85] a tableau-based algorithm was given to check that a system of size m is a model of a formula of size n, with a resulting complexity of $O(m \cdot 2^n)$.

2.1.1 Branching-time *versus* linear-time

The first to study the relative merits of branching vs linear time logic was L. Lamport in [Lam80][6]. Lamport claims that linear time is better for reasoning about concurrent programs, while branching time is better for nondeterministic programs. He was the first to show that linear and branching time logics have incomparable expressive power. We now present his results, using the modern notation of **LTL** and **CTL** (see 2.2 later on for the formal definition of these two logics).

First, Lamport shows that the **CTL** formula $f \doteq \mathbf{EF}p$ is not expressible in **LTL**. The proof consists in defining two structures M_1 and M_2 s.t. no **LTL** formula can distinguish between them, but f is true in one and false in the other. Then, if a **LTL** formula equivalent to f would exist, it would be true in a structure where f is false, a contradiction. Hence f is not expressible in **LTL**. Second, Lamport shows that the **LTL** formula $g \doteq \mathbf{FG}\, p$ has no equivalent **CTL** counterpart, using the same method of above.

Having proved the expressive incomparability of **LTL** and **CTL**, the natural question is: which one is better, if any? A lot of (tentative) answers were given to this question. Lamport argues that in concurrent programs we are interested in correctness along all execution paths, while in nondeterministic programs we are interested in the existence of a successful computation. Then, since the possibility operator **EF** cannot be expressed in **LTL**, the only appropriate logic for nondeterminism is **CTL**. On the other hand, for concurrent programs we need fairness constraints not expressible in **CTL**, then **LTL** is the only choice.

While Lamport deliberately ignored the possibility of constructing a more powerful logic, subsuming both **LTL** and **CTL**, Emerson and Halpern were of different advice [EH86]. They showed that neither **CTL** nor **LTL** was completely adequate for practical applications and they introduced the new branching time logic **CTL***, combining the power of both **CTL** and **LTL**. We will return to **CTL*** in the following Section 2.2. Moreover, in [EH86] some logical difficulties in Lamport's presentation were clarified and a meaningful way of comparing linear- against branching-time specifications was given.

However, **CTL*** is *too much* expressive for practical model-checking purposes. Hence, one line of research aimed at finding adequate extensions of **CTL**, capable of expressing fairness constraints, but retaining an efficient model-checking algorithm. On the one hand, fairness can be introduced into the model. This is the solution of [CES86], where *(weakly) fair Kripke structures* were introduced. The only change to the **CTL** semantics is to interpret path quantifiers over fair paths, rather than all paths. On the other hand, in [EL85] Emerson and Lei proposed **GFCTL**, which is the extension of **CTL** where path quantifiers are interpreted wrt certain (restricted) linear time formulae. Unlike [CES86],

[5]In a letter to A. Prior, 1958.

[6]By the way, [Lam80] became "one of the most frequently cited papers in the temporal-logic literature", to cite Lamport himself. [Lam80] is available from Lamport's home-page, where it is also mentioned that this celebrated paper was rejected two times before being eventually published!

GFCTL can also handle strong fairness. Moreover, Emerson and Lei described a *canonical form* capable of expressing almost all "practical" types of fairness, for which they give a model-checking algorithm of linear complexity wrt the size of the system. Finally, they also showed that for each linear-time logic model-checking algorithm, there is a model-checking algorithm of the same complexity for the least subsuming branching-time logic. Hence, one of the main messages of [EL85] is that, for the purposes of model-checking, branching-time logics are always better than linear-time logics.

Consequently, the first generation of model-checkers (like SMV [McM92b, McM92a] and VIS [Gro96]) used specification formalisms based on branching-time logics, akin to **CTL**. However, the struggle between the branching and the linear paradigms was far from its end. While model-checking algorithms for **CTL** exhibit linear complexity both in the model and in the specification, the model-checking problem for linear-time logics is linear in the model but exponential in the specification. However, in [LP85] Lichtenstein and Pnueli argued that it is the complexity with respect to the model that matters, as specifications are often quite small when compared to the model. Moreover, practitioners often prefer to specify their systems using the more intuitive linear-time formalisms, branching-time logics being unintuitive and hard to use. Hence, linear-time logics evolved and nowadays we have mature linear-time formalisms like ForSpec [AFF$^+$02] and the industrial standard PSL [EF06]. Moreover, in [Var01] several reasons are given in favor of linear-time logic: 1) linear-time logic supports compositional reasoning, which is of fundamental importance when tackling very complex systems made-up from many small components, 2) linear-time logic allows for semi-formal verification and 3) linear-time logic supports explicit as well as symbolic algorithms. Finally, in [NV07] it is argued that branching-time formalisms are even *inadequate* for the purpose of process equivalence, since they distinguish "too much".

We have seen that the branching- and linear-time paradigms have different features. It is fair to say that it is not completely clear if one paradigm definitively prevails against the other.

2.2 The logic CTL*

We now describe the syntax and the semantics of the logic **CTL***, introduced by Emerson and Halpern in [EH86]. We follow the presentation of [CGP00].

Formulae of the logic **CTL*** express properties of *computation trees*. A computation tree is the result of the unwinding of a Kripke structure. Since we require that every state has a successor, the tree is infinite; its root is the starting state of the Kripke structure. **CTL*** is a *branching-time* logic, as it can express the branching nature of computation trees.

There are two kinds of **CTL*** operators: *path quantifiers* and *temporal operators*. Path quantifiers allow reasoning about the branching nature of time and they are interpreted in a given state. In particular, there are two such quantifiers:

- **A** (for all computation paths): means that all computation paths that start in a given state must have some property.
- **E** (for some computation path): means that there is a computation path, starting in a given state, that has some property.

Temporal operators describe properties that hold along computation paths. They are:

- **X** (*ne*X*t time*): some property holds in the next state along this computation path.
- **F** (*in the* F*uture*): some property eventually becomes true in some future state.
- **G** (G*lobally*): some property is true in each state of this computation path.
- **U** (U*ntil*): this operator is more complicated as it combines two properties. It states that the second property must eventually be satisfied and that the first property is true on all preceding states. In this sense, the first property is true *until* the second.
- **R** (R*elease*): it is the logical dual of the previous operator. It states that the second property is true up to and including the first state in which the first property holds. Note that the first property is not required to hold eventually. We can say that the first property *releases* the second.
- **W** (W*eak until*): it is like the until operator, with the difference that the second property is not required to hold eventually. Hence the name.
- **S**: it is the strong version of the release operator. The first property is required to eventually hold.

We now describe the formal syntax and semantics of **CTL***. We have two kinds of formulae: state and path formulae. Intuitively, state formulae are interpreted in a state, while path formulae holds along an entire computation path. Every atomic proposition is a state formula and we can combine state formulae with the usual boolean operators. Moreover, if f is a path formula, then $\mathbf{E}f$ and $\mathbf{A}f$ are state formulae. For path formulae we have that:

- If f is a state formula, then it is also a path formula.
- If f and g are path formulae, then so are $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$, $f\, \mathbf{U}\, g$, $f\, \mathbf{R}\, g$, $f\, \mathbf{W}\, g$ and $f\, \mathbf{S}\, g$.

The semantics of **CTL*** is defined with respect to a *Kripke structure*. A Kripke structure is a triple $\mathcal{M} = (\mathcal{S}, \mathcal{R}, \mathcal{L})$, where $\mathcal{S}$ is a non-empty set of states, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation, which we assume to be total (that is, for all $s \in \mathcal{S}$ there exists $s' \in \mathcal{S}$ s.t. $(s, s') \in \mathcal{R}$), and $\mathcal{L}$ is the labeling function that associates to each state the set of atomic propositions true in that state, i.e., $\mathcal{L} : \mathcal{S} \mapsto 2^{AP}$, where AP is the set of atomic propositions. A *path* is a nonempty, infinite sequence of states: $\pi = s_0, s_1, \ldots$ and $(s_i, s_{i+1}) \in \mathcal{R}$, for each i. (Note that infinite paths exist since $\mathcal{R}$ is total.) With π^i we denote the *suffix* of π starting at state i, i.e., $\pi^i \doteq s_i, s_{i+1}, \ldots$.

We denote paths with π, states with s and atomic propositions with $p \in AP$. Moreover, we use f, f_1, f_2 for state formulae and g, g_1, g_2 for path formulae. When a state formula f is true in state s we write $\mathcal{M}, s \models f$; similarly, for paths π and path formulae g we write $M, \pi \models g$ when g is true along π. (Note that $\models$ has been overloaded.) We define the semantic relation $\models$ by induction on the structure of formulae:

1. $\mathcal{M}, s \models p \iff p \in \mathcal{L}(s)$.
2. $\mathcal{M}, s \models \neg f \iff \mathcal{M}, s \not\models f$.
3. $\mathcal{M}, s \models f_1 \vee f_2 \iff \mathcal{M}, s \models f_1$ or $\mathcal{M}, s \models f_2$.
4. $\mathcal{M}, s \models f_1 \wedge f_2 \iff \mathcal{M}, s \models f_1$ and $\mathcal{M}, s \models f_2$.
5. $\mathcal{M}, s \models \mathbf{A}\, f \iff$ for each π starting at s, $\mathcal{M}, \pi \models f$.
6. $\mathcal{M}, s \models \mathbf{E}\, f \iff$ there exists π starting at s s.t. $\mathcal{M}, \pi \models f$.
7. $\mathcal{M}, \pi \models f \iff s_0$ is the first state of π and $\mathcal{M}, s_0 \models f$.
8. $\mathcal{M}, \pi \models \neg g \iff \mathcal{M}, \pi \not\models g$.
9. $\mathcal{M}, \pi \models g_1 \vee g_2 \iff \mathcal{M}, \pi \models g_1$ or $\mathcal{M}, \pi \models g_2$.
10. $\mathcal{M}, \pi \models g_1 \wedge g_2 \iff \mathcal{M}, \pi \models g_1$ and $\mathcal{M}, \pi \models g_2$.
11. $\mathcal{M}, \pi \models \mathbf{X}\, g \iff \mathcal{M}, \pi^1 \models g$.
12. $\mathcal{M}, \pi \models \mathbf{F}\, g \iff$ there exists $k \geq 0$ s.t. $\mathcal{M}, \pi^k \models g$.
13. $\mathcal{M}, \pi \models \mathbf{G}\, g \iff$ for all $k \geq 0$, $\mathcal{M}, \pi^k \models g$.
14. $\mathcal{M}, \pi \models g_1\, \mathbf{U}\, g_2 \iff$ there exists $k \geq 0$ s.t. $\mathcal{M}, \pi^k \models g_2$, and for each $0 \leq \ j < \ k$, $\mathcal{M}, \pi^j \models g_1$.
15. $\mathcal{M}, \pi \models g_1\, \mathbf{R}\, g_2 \iff$ for each $k \geq 0$, if for each $j < k$ $\mathcal{M}, \pi^j \not\models g_1$, then $\mathcal{M}, \pi^k \models g_2$.
16. $\mathcal{M}, \pi \models g_1\, \mathbf{W}\, g_2 \iff$ for each $k \geq 0$, if for each $j \leq k$ $\mathcal{M}, \pi^j \not\models g_2$, then $\mathcal{M}, \pi^k \models g_1$.
17. $\mathcal{M}, \pi \models g_1\, \mathbf{S}\, g_2 \iff$ there exists $k \geq 0$ s.t. $\mathcal{M}, \pi^k \models g_1$ and for each $0 \leq j \leq k$, $\mathcal{M}, \pi^j \models g_2$.

We say that f_1 and f_2 (resp. g_1 and g_2) are *semantically equivalent*, and we write $f_1 \equiv f_2$ ($g_1 \equiv g_2$), iff for all Kripke structures M and states s (paths π) we have: $M, s \models f_1$ iff $M, s \models f_2$ ($M, \pi \models g_1$ iff $M, \pi \models g_2$, respectively).

It is easy to see that the operators $\neg, \vee, \mathbf{X}, \mathbf{U}$ and $\mathbf{E}$ form a *basis* for **CTL*** formulae, i.e., all **CTL*** formulae can be expressed using only operators in the basis. This follows from the following relations:

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$.
- $\mathbf{A}\, f \equiv \neg \mathbf{E}\, \neg f$.
- $\mathbf{F}\, f \equiv \neg \mathbf{G}\, \neg f$.
- $f\, \mathbf{W}\, g \equiv (f\, \mathbf{U}\, g) \vee \mathbf{G}\, f$.
- $f\, \mathbf{R}\, g \equiv \neg(\neg f\, \mathbf{U}\, \neg g)$.
- $f\, \mathbf{S}\, g \equiv \neg(\neg f\, \mathbf{W}\, \neg g)$.

Other relations are directly derivable from the definitions. Some of them are:

- $f\, \mathbf{U}\, g \equiv (f\, \mathbf{W}\, g) \wedge \mathbf{F}\, g$.
- $f\, \mathbf{R}\, g \equiv (f\, \mathbf{S}\, g) \vee \mathbf{G}\, g$.
- $f\, \mathbf{S}\, g \equiv (f\, \mathbf{R}\, g) \wedge \mathbf{F}\, f$.
- $\mathbf{F}\, f \equiv \mathbf{true}\, \mathbf{U}\, f$.
- $\mathbf{G}\, f \equiv \mathbf{false}\, \mathbf{W}\, f$.
- $f\, \mathbf{U}\, g \equiv g\, \mathbf{S}\, (f \wedge g)$.
- $f\, \mathbf{W}\, g \equiv g\, \mathbf{R}\, (f \wedge g)$.

Moreover, the operators **E**, **A**, **F** e **G** are idempotent, e.g., $\mathbf{E}\,\mathbf{E}\, f \equiv \mathbf{E}\, f$. Also note that **X** distributes on boolean connectives and commutes with negation:

- $\mathbf{X}\,(f \vee g) = \mathbf{X}\, f \vee \mathbf{X}\, g$.
- $\mathbf{X}\,(f \wedge g) = \mathbf{X}\, f \wedge \mathbf{X}\, g$.
- $\mathbf{X}\,(\neg f) = \neg \mathbf{X}\, f$.

Finally, many other temporal operators have been proposed in the literature. We mention here two examples by L. Lamport:

- $f \leadsto g$, read as f *leads to* g [Lam77]. It is equivalent to the **CTL*** formula $\mathbf{A}(f \Rightarrow \mathbf{F}\, g)$, and it means that every occurrence of f is eventually followed by an occurrence of g.
- $f \trianglelefteq g$, meaning that g holds at least as long as f does [Lam83]. It is equivalent to the **CTL*** formula $\mathbf{A}(g\, \mathbf{U}\, \neg f)$.

2.3 The logics CTL and LTL

In this section we will define two noticeable fragments of **CTL***: the branching-time logic **CTL** and the linear-time logic **LTL**. The two logics have incomparable descriptive power, as pointed out before (see 2.1.1, on page 13).

Computation Tree Logic

The logic **CTL** (which stands for *Computation Tree Logic*) is the version of **CTL*** where we cannot directly combine temporal operators. Instead, we require that each of the temporal operators $\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}, \mathbf{W}, \mathbf{S}$ must be immediately preceded by a path quantifier, either **A** or **E**. Formally, **CTL** is the fragment of **CTL*** where the rule for path formulae is:

- If f and g are *state* formulae, then $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$, $f\, \mathbf{U}\, g$, $f\, \mathbf{R}\, g$, $f\, \mathbf{W}\, g$ and $f\, \mathbf{S}\, g$ are path formulae.

In practice, the rule above entails that **CTL** has a total of 14 operators, obtained combining the two path quantifiers **E** and **A** with the seven temporal operators. The resulting modalities are:

$$\mathbf{EX}, \mathbf{AX}, \mathbf{EF}, \mathbf{AF}, \mathbf{EG}, \mathbf{AG}, \mathbf{EU}, \mathbf{AU}, \mathbf{ER}, \mathbf{AR}, \mathbf{EW}, \mathbf{AW}, \mathbf{ES}, \mathbf{AS}\ .$$

This is remarkable, since it shows that **CTL** has a finite number of temporal modalities. This contrasts with **CTL***, which has a (potentially) infinite number of modalities, due to the possibility of arbitrarily nesting path formulae, as shown in [RM00].

Some typical **CTL** formulae are:

- $\mathbf{AG}f$: *invariance*, f holds in every (reachable) state.
- $\mathbf{AG}(f \Rightarrow \mathbf{AF}g)$: *temporal implication*, whenever f holds, it is eventually followed by g.
- $\mathbf{AG}(f \Rightarrow \mathbf{EF}g)$: any occurrence of f can be followed by an occurrence of g.

One of the main limitation of **CTL** is that it cannot express many fairness properties (see 2.1.1, on page 13).

All **CTL** operators can be expressed as a combination of operators in the following basis: $\neg, \vee, \mathbf{EX}, \mathbf{EG}, \mathbf{EU}$[7]. In fact, the following relations hold:

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$.
- $\mathbf{AX}f \equiv \neg\mathbf{EX}\neg f$.
- $\mathbf{EF}f \equiv \mathbf{E}(\mathbf{true}\, \mathbf{U}\, f)$.

[7]The choice of this basis is not random. As we will see when extending **CTL** in Section 4.2, the fact that all 14 **CTL** temporal operators can be expressed, basically, from **EG** and **EU** allows us to extend **CTL** in a very natural way.

- $\mathbf{AG}f \equiv \neg\mathbf{EF}\neg f$.
- $\mathbf{AF}f \equiv \neg\mathbf{EG}\neg f$.
- $\mathbf{E}(f\,\mathbf{W}\,g) \equiv \mathbf{E}(f\,\mathbf{U}\,g) \vee \mathbf{EG}f$.
- $\mathbf{E}(f\,\mathbf{R}\,g) \equiv \mathbf{E}(g\,\mathbf{W}\,f \wedge g)$
- $\mathbf{E}(f\,\mathbf{S}\,g) \equiv \mathbf{E}(g\,\mathbf{U}\,f \wedge g)$.
- $\mathbf{A}(f\,\mathbf{U}\,g) \equiv \neg\mathbf{E}(\neg f\,\mathbf{R}\,\neg g)$.
- $\mathbf{A}(f\,\mathbf{W}\,g) \equiv \neg\mathbf{E}(\neg f\,\mathbf{S}\,\neg g)$.
- $\mathbf{A}(f\,\mathbf{R}\,g) \equiv \neg\mathbf{E}(\neg f\,\mathbf{U}\,\neg g)$.
- $\mathbf{A}(f\,\mathbf{S}\,g) \equiv \neg\mathbf{E}(\neg f\,\mathbf{W}\,\neg g)$.

As noted in [Lar95], it is quite surprising that, while on the one hand **AU** can be defined from **EU** and **EG** as

$$\mathbf{A}(f\,\mathbf{U}\,g) \equiv \neg\mathbf{E}(\neg g\,\mathbf{U}\,\neg f \wedge \neg g) \wedge \neg\mathbf{EG}\neg g \ ,$$

on the other hand **EU** *cannot* be defined in terms of **AU** and **EF**. This is an curious expressibility result about what are sometimes called ***CTL** combinators*.

Linear Temporal Logic

The logic **LTL** is the fragment of **CTL*** consisting of formulae of the form $\mathbf{A}\,g$, for g an **LTL** path formula. The only allowed path quantifier in a **LTL** formula is the starting **A**. This is why **LTL** is *linear*: as path formulae are interpreted along all paths, **LTL** cannot describe the branching nature of time. Formally, **LTL** path formulae are defined as follows (g, g_1 and g_2 are path formulae):

- $p \in AP$ is a path formula.
- $\neg g$, $g_1 \vee g_2$ e $g_1 \wedge g_2$ are path formulae;
- $\mathbf{X}\,g$, $\mathbf{F}\,g$, $\mathbf{G}\,g$, $g_1\,\mathbf{U}\,g_2$, $g_1\,\mathbf{W}\,g_2$, $g_1\,\mathbf{R}\,g_2$ and $g_1\,\mathbf{S}\,g_2$ are path formulae.

LTL can express general fairness constraints. Let g, g_1, g_2 be path formulae. Referring to the kind of fairness constraints previously introduced (see 1.4, on page 6), we have:

- *Impartiality*, $\mathbf{A}\,\mathbf{GF}\,g$: g is true infinitely often.
- *Weak fairness*, $\mathbf{A}\,(\mathbf{FG}\,g_1 \Rightarrow \mathbf{GF}\,g_2)$: if g_1 holds almost everywhere, then g_2 holds infinitely often. This formula can be somewhat simplified (omitting the initial **A** quantifier):

$$\begin{aligned}\mathbf{FG}\,g_1 \Rightarrow \mathbf{GF}\,g_2 &\equiv (\mathbf{GF}\,\neg g_1) \vee (\mathbf{GF}\,g_2)\\ &\equiv \mathbf{GF}\,(\neg g_1 \vee g_2)\\ &\equiv \mathbf{GF}\,(g_1 \Rightarrow g_2)\ ,\end{aligned}$$

where the second line follows from the distributive property of the infinitary operator "**GF**" over $\vee$. So, the weak fairness of g_2 with respect to g_1 is equivalent to the impartiality of $g_1 \Rightarrow g_2$. Finally, note that the fact that all paths are (weakly) fair can be expressed also in **CTL**, as $\mathbf{AGAF}(g_1 \Rightarrow g_2)$. However, this is of quite little interest, as fairness formulae are generally of the form **A** (fairness constraints $\Rightarrow$ property), and, in general, those formulae are not expressible in **CTL**.

- *Strong fairness*, $\mathbf{A}\,(\mathbf{GF}\,g_1 \Rightarrow \mathbf{GF}\,g_2)$: if g_1 holds infinitely often, then g_2 holds infinitely often.

In the next section, we will present an expressive temporal logic, subsuming **CTL***, and hence both **CTL** and **LTL**.

2.4 The branching-time propositional μ-calculus

The μ-calculus is a powerful branching-time temporal formalism for expressing properties of transition systems. It is based on least and greatest fixpoint operators and it is so expressive that most of current temporal logics have direct translations into the μ-calculus. Moreover, it has a simple syntax and its semantics is based on a well-understood theorem of Tarski. Efficient algorithms have been developed for the full μ-calculus and for its most useful fragments. Despite its theoretical interest, μ-calculus is not so widespreadly used by practitioners, as its formulae can become quickly unintelligible. For these reasons, its has been called the "machine-level" temporal logic [May97].

We will present the μ-calculus in the version of D. Kozen [Koz83]. As our Kripke structures are *endogenous*, i.e., transitions are unnamed, we will slightlymodify the syntax and use the "anonymous" next-state operators **EX** and **AX**, instead of the traditional modalities $\Diamond$ and $\Box$. All μ-calculus formulae are state formulae and are interpreted over some Kripke structure $\mathcal{M} = (\mathcal{S}, \mathcal{R}, \mathcal{L})$. Let $V = Q_1, Q_2, \ldots$ be a set of *relational variables*; intuitively, each relational variable can hold a subset of $\mathcal{S}$. The set of μ-calculus formulae is the least set obeying the following syntactic rules:

- Each atomic proposition $p \in AP$ is a formula.
- Each relational variable $Q \in V$ is a formula.
- If f and g are formulae, then so is $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{EX}f$ and $\mathbf{AX}f$.
- If $Q \in V$ and f is a formula, then $\mu\, Q.\ f$ and $\nu\, Q.\ f$ are formulae, provided that f is *syntactically monotone* in Q, i.e., Q occurs under an even number of negations in f.

As usual, each relational variable may be *free*, or *bound* by a fixpoint operator. Sometimes we write $f(Q)$ to emphasize that Q is free in f. *Closed* formulae are those with no free variables. We need formulae to be monotone in their free variables to ensure the existence of fixpoints.

Intuitively, a closed μ-calculus formula f denotes a set of states, namely the set of states in which f is true. We interpret formulae with respect to a Kripke structure $\mathcal{M}$ and environment e, and write $[\![f]\!]^e_{\mathcal{M}}$ for the set of states in which f holds. An *environment* e is a map $e : V \mapsto 2^{\mathcal{S}}$ assigning a subset of $\mathcal{S}$ to each relational variable. With $e[Q \leftarrow W]$, $W \subseteq \mathcal{S}$, we denote the new environment which is equal to e, except returning W on Q, formally:

$$e[Q \leftarrow W](Q') = \begin{cases} e(Q') & \text{if } Q \neq Q' \\ W & \text{if } Q = Q' \ . \end{cases}$$

We can see a formula $f(Q_1, \ldots, Q_n)$ with n free variables $Q_1, \ldots, Q_n$ as a *predicate transformer* $\tau : 2^{\mathcal{S}^n} \mapsto 2^{\mathcal{S}^n}$, defined as

$$\tau(W_1, \ldots, W_n) \doteq [\![f]\!]^{e'}_{\mathcal{M}} \ , \ e' \doteq e[Q_1 \leftarrow W_1] \cdots [Q_n \leftarrow W_n] \ ,$$

which transforms subsets of $\mathcal{S}^n$ into (possibly different) subsets of $\mathcal{S}^n$. We will indicate with $\Pi(f)$ the predicate transformer denoted by f. If f is a closed formula, i.e., $n = 0$, then it denotes the constant predicate transformer $\Pi(f) = [\![f]\!]^e_{\mathcal{M}}$.

$[\![f]\!]^e_{\mathcal{M}}$ is defined with the following structural induction on formulae:

- $[\![p]\!]^e_{\mathcal{M}} = \{\ s \in \mathcal{S} \ \mid p \in \mathcal{L}(s)\ \}$.
- $[\![Q]\!]^e_{\mathcal{M}} = e(Q)$.
- $[\![\neg f]\!]^e_{\mathcal{M}} = \mathcal{S} \setminus [\![f]\!]^e_{\mathcal{M}}$.
- $[\![f \wedge g]\!]^e_{\mathcal{M}} = [\![f]\!]^e_{\mathcal{M}} \cap [\![g]\!]^e_{\mathcal{M}}$.
- $[\![f \vee g]\!]^e_{\mathcal{M}} = [\![f]\!]^e_{\mathcal{M}} \cup [\![g]\!]^e_{\mathcal{M}}$.
- $[\![\mathbf{EX}f]\!]^e_{\mathcal{M}} = \{\ s \ \mid \exists s' \text{ s.t. } (s, s') \in \mathcal{R} \text{ and } s' \in [\![f]\!]^e_{\mathcal{M}}\ \}$.
- $[\![\mathbf{AX}f]\!]^e_{\mathcal{M}} = \{\ s \ \mid \forall s' \ (s, s') \in \mathcal{R} \text{ implies } s' \in [\![f]\!]^e_{\mathcal{M}}\ \}$.
- $[\![\mu Q.\ f(Q)]\!]^e_{\mathcal{M}} = \mu W.\ \tau(W)$, and
  $[\![\nu Q.\ f(Q)]\!]^e_{\mathcal{M}} = \nu W.\ \tau(W)$, where $\tau(W) \doteq \Pi(f)$.

The above (syntactic) monotonicity requirement ensures that formulae denote only *monotone* predicate transformers. ($\tau(W)$ is monotone iff $W_1 \subseteq W_2$ implies $\tau(W_1) \subseteq \tau(W_2)$). Then, applying the Knaster-Tarski theorem (see 1.5, on page 7) to the complete lattice $\langle \mathcal{S}, \subseteq \rangle$, we have that such predicate transformers have maximal and minimal fixpoints, and the semantics of μ-calculus is well-defined.

Formally, all introduced operators are monotone, except negation. But we can push negations inside formulae, using usual dualities, until we reach atomic propositions or relational variables. (Applying this procedure, we would eventually obtain a formula in *positive normal form*, that is a formula where the only negations appear in front of atomic formulae.) Then, if all relational variables are under an even number of negations, they will be negation-free after this process, and all formulae will be monotonic. That all operators except negation are monotone is what is stated in the following

Theorem 2.1. *The operators $\wedge, \vee, \mathbf{EX}, \mathbf{AX}, \mu, \nu$ preserve monotonicity.*

Proof. We proceed by cases.

- $f \equiv f_1 \wedge f_2$. Assume, without loss of generality, that f_1 and f_2 contain only one free variable. Let $\tau_1(W) \doteq \Pi(f_1)$ and $\tau_2(W) \doteq \Pi(f_2)$. Then $\tau \doteq \Pi(f) = \tau_1 \cap \tau_2$. Now, assume that $W_1 \subseteq W_2$. Then, $\tau_1(W_1) \subseteq \tau_1(W_2)$ and $\tau_2(W_1) \subseteq \tau_2(W_2)$, by induction hypothesis. It follows that
$$\tau(W_1) \doteq \tau_1(W_1) \cap \tau_2(W_1) \subseteq \tau_1(W_2) \cap \tau_2(W_2) =: \tau(W_2) \ .$$
- $f \equiv f_1 \vee f_2$. Like the previous case.
- $f \equiv \mathbf{EX}f'$. $\tau(W) \doteq \Pi(f) = \{ \ s \mid \exists s' \text{ s.t. } (s, s') \in \mathcal{R} \text{ and } s' \in \tau'(W) \ \}$, with $\tau' \doteq \Pi(f')$. Since τ' is monotone by induction hypothesis, the monotonicity of τ easily follows from an argument like above.
- $f \equiv \mathbf{AX}f'$. Like the previous case.
- $f \equiv \mu Q.\ g(Q)$. (Note that g may have other free variables besides Q). This case is more involved. Without loss of generality, assume that g has exactly two free variables, Q and R. Let $\gamma(W, V) \doteq \Pi(g)$. By definition and the Tarski's theorem, $\tau(V) \doteq \Pi(f) = \bigcap Pre_V$, where $Pre_V \doteq \{ \ W \mid \gamma(W, V) \subseteq W \ \}$ are the prefixpoints of γ wrt V. Now assume $V_1 \subseteq V_2$. Then for all prefixpoints W wrt V_2, $\gamma(W, V_1) \subseteq \gamma(W, V_2) \subseteq W$, the first inequality holding by induction hypothesis. Hence, all prefixpoints W wrt V_2 are also prefixpoints wrt V_1, that is $Pre_{V_2} \subseteq Pre_{V_1}$. Hence $\bigcap Pre_{V_1} \subseteq \bigcap Pre_{V_2}$, as the glb of Pre_{V_2} cannot be less than the glb of Pre_{V_1}. Then $\tau(V_1) \subseteq \tau(V_2)$, as required.
- $f \equiv \nu Q.\ g(Q)$. Just like the previous case. Only note that in this case we have $Post_{V_1} \subseteq Post_{V_2}$ and $\bigcup Post_{V_1} \subseteq \bigcup Post_{V_2}$.

□

Moreover, if a formula is in positive normal form (from now on, pnf), then there are cases in which it also preserves unions and intersections, as stated in the following two theorems.

Theorem 2.2. *Let $f(X)$ be a formula in pnf built from $\wedge, \vee, \mathbf{EX}, \nu$ and arbitrary* closed *formulae. Moreover, assume that whenever $\wedge$ is used, then at least one of its two operands is a closed formula. Hence, $\tau(W) \doteq \Pi(f)$ (the predicate transformer denoted by $f(X)$) satisfies*
$$\tau(A \cup B) = \tau(A) \cup \tau(B) \ , \tag{2.1}$$
for all $A, B \subseteq \mathcal{S}$.

Proof. If f is a closed formula, equation 2.1 is trivial. For the other cases we have:

- $f \equiv f_1 \wedge f_2$. We know that at most one of f_1, f_2 is non-closed. Without loss of generality, assume that $f_1(X)$ depends on X and that f_2 is closed. Let $\tau_1(W) \doteq \Pi(f_1)$, $B \doteq \Pi(f_2)$ and $\tau(W) \doteq \Pi(f) = \tau_1(W) \cap B$. Then, for all $A_1, A_2 \subseteq \mathcal{S}$,

$$\begin{aligned}
\tau(A_1 \cup A_2) &= \tau_1(A_1 \cup A_2) \cap B = \text{(by induction hyp.)} && (2.2)\\
&= (\tau_1(A_1) \cup \tau_1(A_2)) \cap B = && (2.3)\\
&= (\tau_1(A_1) \cap B) \cup (\tau_1(A_2) \cap B) = && (2.4)\\
&= \tau(A_1) \cup \tau(A_2)\ , && (2.5)
\end{aligned}$$

hence $\tau(A_1 \cup A_2) = \tau(A_1) \cup \tau(A_2)$.

- $f \equiv f_1 \vee f_2$. Let $\tau_1(W) \doteq \Pi(f_1)$, $\tau_2(W) \doteq \Pi(f_2)$ and $\Pi(f) = \tau(W) \doteq \tau_1(W) \cup \tau_2(W)$. Then, for all $A, B \subseteq \mathcal{S}$,

$$\begin{aligned}
\tau(A \cup B) &= \tau_1(A \cup B) \cup \tau_2(A \cup B) = \text{(by induction hyp.)} && (2.6)\\
&= (\tau_1(A) \cup \tau_1(B)) \cup (\tau_2(A) \cup \tau_2(B)) = && (2.7)\\
&= (\tau_1(A) \cup \tau_2(A)) \cup (\tau_1(B) \cup \tau_2(B)) = && (2.8)\\
&= \tau(A) \cup \tau(B)\ , && (2.9)
\end{aligned}$$

hence $\tau(A \cup B) = \tau(A) \cup \tau(B)$.

- $f \equiv \mathbf{EX}f'$. Let $\tau(W) \doteq \Pi(f) = \{\ s \mid \exists s' \text{ s.t. } P(s, s', \tau'(W))\ \}$, where $\tau' \doteq \Pi(f')$ and

$$P(s, s', A) :\equiv (s, s') \in \mathcal{R} \text{ and } s' \in A\ .$$

Then, for all $A_1, A_2 \subseteq \mathcal{S}$, we have that:

$$\begin{aligned}
\tau(A \cup B) &= \{\ s \mid \exists s' \text{ s.t. } P(s, s', \tau'(A \cup B))\ \} = \text{(by induction hyp.)}\\
&= \{\ s \mid \exists s' \text{ s.t. } P(s, s', \tau'(A) \cup \tau'(B))\ \} =\\
&= \{\ s \mid \exists s' \text{ s.t. } (P(s, s', \tau'(A)) \text{ or } P(s, s', \tau'(B)))\ \} =\\
&= \{\ s \mid (\exists s' \text{ s.t. } P(s, s', \tau'(A))) \text{ or } (\exists s' \text{ s.t. } P(s, s', \tau'(B)))\ \} =\\
&= \{\ s \mid \exists s' \text{ s.t. } P(s, s', \tau'(A))\ \} \cup \{\ s \mid \exists s' \text{ s.t. } P(s, s', \tau'(B))\ \} =\\
&= \tau(A) \cup \tau(B)\ ,
\end{aligned}$$

hence $\tau(A \cup B) = \tau(A) \cup \tau(B)$.

- $f \equiv \nu Q.\ g(Q)$. Let $\gamma(W, V) \doteq \Pi(g)$. Then, by definition and the Tarski's theorem, $\tau(V) \doteq \Pi(f) = \bigcup Post_V$, where

$$Post_V \doteq \{\ W \mid W \subseteq \gamma(W, V)\ \}$$

are the postfixpoint of γ wrt V. Then, for all $A_1, A_2 \subseteq \mathcal{S}$, we have that:

$$\tau(A \cup B) = \bigcup Post_{A \cup B} =$$

$$\begin{aligned}
&= \bigcup \{\ W \ \mid W \subseteq \gamma(W, A \cup B)\ \} = \text{(by induction hyp.)} \\
&= \bigcup \{\ W \ \mid W \subseteq \gamma(W, A) \cup \gamma(W, B)\ \} = \\
&= \bigcup \{\ W \ \mid W \subseteq \gamma(W, A) \text{ or } W \subseteq \gamma(W, B)\ \} = \\
&= (\bigcup \{\ W \ \mid W \subseteq \gamma(W, A)\ \}) \cup (\bigcup \{\ W \ \mid W \subseteq \gamma(W, B)\ \}) \\
&= \tau(A) \cup \tau(B) \ ,
\end{aligned}$$

hence $\tau(A \cup B) = \tau(A) \cup \tau(B)$, and this concludes the proof.

□

Obviously, there is a dual version of theorem 2.2, which we state without proof.

Theorem 2.3. *Let $f(X)$ be a formula in pnf built from $\wedge, \vee, \mathbf{AX}, \mu$ and arbitrary* closed *formulae. Moreover, assume that whether $\vee$ is used, then at least one of its two operands is a closed formula. Hence, $\tau(W) \doteq \Pi(f)$ (the predicate transformer denoted by $f(X)$) satisfies*

$$\tau(A \cap B) = \tau(A) \cap \tau(B) \ , \tag{2.10}$$

for all $A, B \subseteq \mathcal{S}$.

The importance of theorems 2.2 and 2.3 will be shown in Chapter 6 (see 6.1.2, on page 70), where we will see that, whenever they hold, they allow a more efficient representation of approximants, which ultimately leads to an substantial improvement in execution-speed.

Now, let us prove that the least fixpoint operator μ is the dual of the greatest fixpoint operator ν.

Theorem 2.4. *If $f_1 \doteq \mu\, Q.\ f(Q)$ is a valid formula, then $f_2 \doteq \neg\nu\, Q.\ \neg f(\neg Q)$ is a valid formula and $[\![f_1]\!]^e_M = [\![f_2]\!]^e_M$.*

Proof. To show that f_2 is a valid formula it is sufficient to note that, if each occurrence of Q in f is under an even number of negations, so are occurrences of Q in $\neg f(\neg Q)$. So f_2 is a valid formula.

In the following chain of equalities, the first term is equal to $[\![f_2]\!]^e_M$ and the last to $[\![f_1]\!]^e_M$:

$$\begin{aligned}
[\![\neg\nu\, Q.\ \neg f(\neg Q)]\!]^e_M &= S \setminus \bigcup \left\{\ W \ \middle|\ W \subseteq (S \setminus [\![f]\!]^{e[Q \leftarrow S \setminus W]}_M)\ \right\} = \\
&= \bigcap \left\{\ S \setminus W \ \middle|\ W \subseteq (S \setminus [\![f]\!]^{e[Q \leftarrow S \setminus W]}_M)\ \right\} = \\
&= \bigcap \left\{\ S \setminus W \ \middle|\ [\![f]\!]^{e[Q \leftarrow S \setminus W]}_M \subseteq S \setminus W\ \right\} = \\
&= \bigcap \left\{\ W \ \middle|\ [\![f]\!]^{e[Q \leftarrow W]}_M \subseteq W\ \right\} .
\end{aligned}$$

□

CTL formula	Corresponding μ-calculus formula
$\mathbf{EF}f$	$\mu Q.\ f \vee \mathbf{EX}Q$
$\mathbf{AF}f$	$\mu Q.\ f \vee \mathbf{AX}Q$
$\mathbf{EG}f$	$\nu Q.\ f \wedge \mathbf{EX}Q$
$\mathbf{AG}f$	$\nu Q.\ f \wedge \mathbf{AX}Q$
$\mathbf{E}(f\,\mathbf{U}\,g)$	$\mu Q.\ g \vee (f \wedge \mathbf{EX}Q)$
$\mathbf{A}(f\,\mathbf{U}\,g)$	$\mu Q.\ g \vee (f \wedge \mathbf{AX}Q)$
$\mathbf{E}(f\,\mathbf{R}\,g)$	$\nu Q.\ g \wedge (f \vee \mathbf{EX}Q)$
$\mathbf{A}(f\,\mathbf{R}\,g)$	$\nu Q.\ g \wedge (f \vee \mathbf{AX}Q)$
$\mathbf{E}(f\,\mathbf{S}\,g)$	$\mu Q.\ g \wedge (f \vee \mathbf{EX}Q)$
$\mathbf{A}(f\,\mathbf{S}\,g)$	$\mu Q.\ g \wedge (f \vee \mathbf{AX}Q)$
$\mathbf{E}(f\,\mathbf{W}\,g)$	$\nu Q.\ g \vee (f \wedge \mathbf{EX}Q)$
$\mathbf{A}(f\,\mathbf{W}\,g)$	$\nu Q.\ g \vee (f \wedge \mathbf{AX}Q)$

Table 2.1: Translating **CTL** formulae into μ-calculus.

Obviously, with an analogous proof, also the converse relation $\nu Q.\ f(Q) \equiv \neg\mu Q.\ \neg f(\neg Q)$ can be established (where $\equiv$ denotes semantical equivalence).

Methods for μ-calculus model-checking will be presented in Chapter 3 and in Chapter 5.

2.5 Translating logics into μ-calculus

Translating CTL

The logic **CTL** admits a succinct and elegant translation into the μ-calculus, being the temporal modalities of **CTL** easily expressible as fixpoints of simple predicate transformers. As we let the operators **EX** and **AX** be common to both logics, we do not consider them here. The translation is shown in table 2.1. It is easy to check that all **CTL** dualities hold, only using the usual De Morgan's laws and μ/ν dualities.

Translating CTL*

It is also possible to translate the whole **CTL*** into μ-calculus [Dam94]. However, the translation is neither succinct, nor conceptually simple. The method relies on various tableau constructions and requires time that is doubly exponential in the length of the **CTL*** formula in the worst case. Moreover, this complexity seems to be unavoidable.

In the next chapter we will see a basic model-checking method for the μ-calculus.

Chapter 3

Symbolic model-checking

In this chapter we present the symbolic model-checking approach to formal verification, concentrating on properties written in μ-calculus formulae. In Section 3.1 we present the simplest known algorithm for model-checking the μ-calculus, based on a theorem by Kleene. In Section 3.2 we present ROBDDs, a data structure that allows for the symbolic representation of Kripke structures. In Section 3.3 we show how ROBDDs can be used with the simple iterative algorithm of Section 3.1. In Section 3.4 we deal with complexity considerations on both the symbolic and explicit approaches to μ-calculus model-checking. Finally, in Section 3.5 we present a practical example of formal verification with NuSMV, a symbolic model-checker of industrial strength.

3.1 Simple μ-calculus model-checking

When defining the semantics of μ-calculus expressions, we used the Knaster-Tarski theorem on the lattice $\langle \mathcal{S}, \subseteq \rangle$ to characterize maximal and minimal fixpoints of monotone predicate transformers (see 1.5, on page 7 for the Knaster-Tarski theorem and see 2.4, on page 20 for the semantics of μ-calculus). In that setting, we saw that

$$\begin{aligned} \mu W.\, \tau(W) &= \bigcap \{\ W \mid \tau(W) \subseteq W\ \}, \quad \text{and} \\ \nu W.\, \tau(W) &= \bigcup \{\ W \mid W \subseteq \tau(W)\ \}\ . \end{aligned}$$

Unfortunately, while this property is satisfying for the semantical characterization of the μ-calculus, it is of little use in practical model-checking. The key idea came from Kleene [Kle52], who gave an iterative method for computing least and greatest fixpoints, as we now explain.

[0]W. Bolyai to his son Johann in urging him to claim the invention of non-Euclidean geometry without delay.

First, we need some definitions. Consider a complete lattice $\langle A, \sqsubseteq \rangle$ and a monotonic function $f : A \mapsto A$. We extend f on subsets $B \subseteq A$, defining

$$f^*(B) \doteq \{\ f(x) \ \mid x \in B\ \}\ . \tag{3.1}$$

When it is clear from the context, we shall write f instead of its extension f^*. We define the iterates of f as usual: $f^0 = Id_A$, $f^{n+1} = f \circ f^n$, where $\circ$ denotes composition of functions and Id_A is the identity function on A, defined as $\{\ (a, a) \ \mid a \in A\ \}$.

A subset $B = \{b_0, b_1, \ldots\}$ of A is an *increasing chain* if its elements satisfy:

$$b_0 \sqsubseteq b_1 \sqsubseteq \cdots\ , \tag{3.2}$$

Decreasing chains are defined similarly. We say that f is $\sqcap$-*continuous* if it commutes with $\sqcap$ on decreasing chains, i.e., for each decreasing chain $B = \{b_0, b_1, \ldots\}$, $\sqcap f(B) = f(\sqcap B)$. Analogously, f is $\sqcup$-*continuous* if $\sqcup f(B) = f(\sqcup B)$ on increasing chains B.

Remark. *If f is monotone, we get $f(\sqcap B) \sqsubseteq \sqcap f(B)$ and $\sqcup f(B) \sqsubseteq f(\sqcup B)$. In fact, note that for all $x \in B$, $\sqcap B \sqsubseteq x$, hence $f(\sqcap B) \sqsubseteq f(x)$. Since this is true for all x's, then $f(\sqcap B) \sqsubseteq \sqcap_{x\in B} f(x) = \sqcap f(B)$. We ha proved that monotonicity implies*

$$f(\bigsqcap B) \sqsubseteq \bigsqcap f(B)\ . \tag{3.3}$$

Clearly, with an analogous argument for lubs, we have

$$\bigsqcup f(B) \sqsubseteq f(\bigsqcup B)\ . \tag{3.4}$$

We now have the following

Theorem 3.1 (Kleene, 1952)**.** *Let $\langle A, \sqsubseteq \rangle$ be a complete lattice and let f be a monotonic function $f : A \mapsto A$. Then:*

- *If f is $\sqcap$-continuous,*

$$\nu x.\ f(x) = \bigsqcap_{i\geq 0} f^i(\top)\ . \tag{3.5}$$

- *If f is $\sqcup$-continuous,*

$$\mu x.\ f(x) = \bigsqcup_{i\geq 0} f^i(\bot)\ . \tag{3.6}$$

Proof. We only prove the first point, the second being similar. Assume f is $\sqcap$-continuous and define $d \doteq \sqcap_{i\geq 0} f^i(\top)$. Then:

$$f(d) \quad = \quad f(\bigsqcap_{i\geq 0} f^i(\top))$$

$$\begin{aligned}
&= \bigsqcap_{i\geq 0} f^{i+1}(\top) \\
&= \bigsqcap_{i\geq 0} f^{i+1}(\top) \sqcap \top \\
&= \bigsqcap_{i\geq 0} f^{i+1}(\top) \sqcap f^{0}(\top) \\
&= \bigsqcap_{i\geq 0} f^{i}(\top) \\
&= d \ ,
\end{aligned}$$

that is, d is a fixpoint of f. For each postfixpoint $x \sqsubseteq f(x)$, we have that $x \sqsubseteq \top$, and applying f once,

$$x \sqsubseteq f(x) \sqsubseteq f(\top) \ .$$

Iterating, we have that for all i:

$$x \sqsubseteq f^{i}(\top) \ ,$$

hence all $f^{i}(\top)$ are upper bounds on the set of postfixpoints of f. Then taking the least one of these upper bounds, we obtain

$$x \sqsubseteq \bigsqcap_{i\geq 0} f^{i}(\top) = d \ ,$$

i.e., d is bigger than all postfixpoint. Now, being d a fixpoint, and hence a postfixpoint, we have that d is also the greatest postfixpoint:

$$d = \bigsqcup \{\ x \ \mid x \sqsubseteq f(x)\ \} = \nu\, x.\ f(x) \ ,$$

which concludes the proof. □

If the lattice is *finite*, i.e., has a finite number of elements, then every monotonic function is also both $\bigsqcap$ and $\bigsqcup$-continuous. In fact, in finite lattices every (increasing or decreasing) chain $B = \{b_0, b_1, \ldots\}$ has a finite number of elements and its limit (lub or glb) its an element of the chain itself, namely the "last" element. More formally, if $B = \{b_0, b_1, \ldots, b_n\}$ is a decreasing (increasing, resp.) chain with n elements, then

$$\bigsqcap B = b_n \quad \Big(\bigsqcup B = b_n, \text{ resp.}\Big) \ .$$

Now, it is easy to see that, by monotonicity, also $\{f(b_0), f(b_1), \ldots, f(b_n)\}$ is a finite decreasing (increasing, resp.) chain, hence

$$\bigsqcap f(B) = f(b_n) \quad \Big(\bigsqcup f(B) = f(b_n), \text{ resp.}\Big) \ .$$

We have just shown that $f(\bigsqcap B) = \bigsqcap f(B)$ ($f(\bigsqcup B) = \bigsqcup f(B)$, resp.), that is f is continuous.

Algorithm 3.1: Naive evaluation of $\mu P.\ \tau(P)$.

```
  input  : A continuous function $\tau$
  output: $\mu\ \tau$
1 $Q' := \emptyset$
2 repeat
3     $Q := Q'$
4     $Q' := \tau(Q)$
5 until $Q \neq Q'$
6 return $Q$
```

Algorithm 3.2: Naive evaluation of $\nu P.\ \tau(P)$.

```
  input  : A continuous function $\tau$
  output: $\nu\ \tau$
1 $Q' := \mathcal{S}$
2 repeat
3     $Q := Q'$
4     $Q' := \tau(Q)$
5 until $Q \neq Q'$
6 return $Q$
```

Figure 3.1: Simple iterative evaluation of fixpoints.

We can apply Kleene's theorem to the complete lattice $\langle \mathcal{S}, \subseteq \rangle$ and monotonic predicate transformers τ, which are also continuous as $\mathcal{S}$ has a finite number of elements, say n. For example, in order to compute the least fixpoint of the μ-calculus formula $f(Q)$, we apply an iterative procedure building the increasing chain $B = \{A_i\}$, as follows:

$$\begin{aligned}
&A_0 = \emptyset \\
&A_1 = [\![f]\!]_M^{e[Q\leftarrow A_0]} \\
&A_2 = [\![f]\!]_M^{e[Q\leftarrow A_1]} \\
&\vdots \\
&A_{i+1} = [\![f]\!]_M^{e[Q\leftarrow A_i]} \\
&\vdots
\end{aligned}$$

As $\mathcal{S}$ is finite, we cannot have more than $n+1$ consecutive increases. Hence, surely $A_{n+1} = A_n$ and the algorithm terminates. However, in general convergence is achieved in fewer iterations, as we detect $A_{j+1} = A_j$. So, we obtain the algorithm of figure 3.1.

3.2 Data-structures for model-checking

Symbolic model-checking gained popularity both for its easy to understand algorithms and for the existence of efficient data-structures for symbolically represent boolean functions. In this section we introduce ROBDDs (Reduced Ordered Binary Decision Diagram, introduced in [Bry86]), a canonical and efficient way of representing boolean functions, and we show how we can do model-checking with ROBDDs.

Let us start with an example[1]. Consider the *parity function*

$$f(x_0, x_1, x_2) \doteq x_0 \oplus x_1 \oplus x_2 \ . \tag{3.7}$$

Given a total order on variables, we can build the *decision tree* T_f associated to f. Intuitively, a decision tree is a rooted, directed tree, whose non-terminal nodes represent variables and whose leaves are either 0 or 1. Each non-terminal node x has two successors, the left one being associated with the value $x = 0$ and the right with $x = 1$. A depth first traversal of T_f correspond to an assignment of values for the x_i's (appearing in T_f in the specified order), and the value at the leaf is the corresponding value of the boolean function. In our example, suppose $x_0 < x_1 < x_2$ is an ordering of variables. Then the corresponding T_f is shown in figure 3.2. For example, the path corresponding to $x_0 = 0, x_1 = 1, x_2 = 1$ goes from the root to 0, then $f(0, 1, 1) = 0$.

We can immediately note that in T_f there is a lot of redundancy. In fact, the two subtrees which we get following $x_0 = 0, x_1 = 1$ and $x_0 = 1, x_1 = 0$, resp., are isomorphic. And this is true for many other paths. We can share this redundant information by removing one copy of the two (or more) identical subtrees and adding arcs accordingly. The synthetic version of T_f is shown in figure 3.3 and it is called a *decision diagram*.

What Bryant did in its seminal work on ROBDDs [Bry86] was to spot systematic algorithms for reducing and manipulating such data structures. Reduction aims at finding a *canonical form* for a given formula and variable ordering. Canonical forms are important since they allow checking equality of two formulae and satisfiability on constant time. Manipulation of ROBDDs means finding fast algorithms for usual boolean operations on formulae.

In particular, consider boolean functions on n boolean variables, i.e., of the form $f(x_1, \ldots, x_n)$. With ROBDD(f) we denote the ROBDD associated to f and with $|\text{ROBDD}(f)|$ its size (measured as number of nodes). Then, it can be shown that there are algorithms s.t.:

- Given ROBDD(f) and ROBDD(g), ROBDD($f \vee g$) and ROBDD($f \wedge g$) can be computed in time $O(|\text{ROBDD}(f)| \cdot |\text{ROBDD}(g)|)$.
- Given ROBDD(f), ROBDD($\neg f$) can be computed in time proportional to $O(|\text{ROBDD}(f)|)$.

[1]The following is taken from [BCJM96].

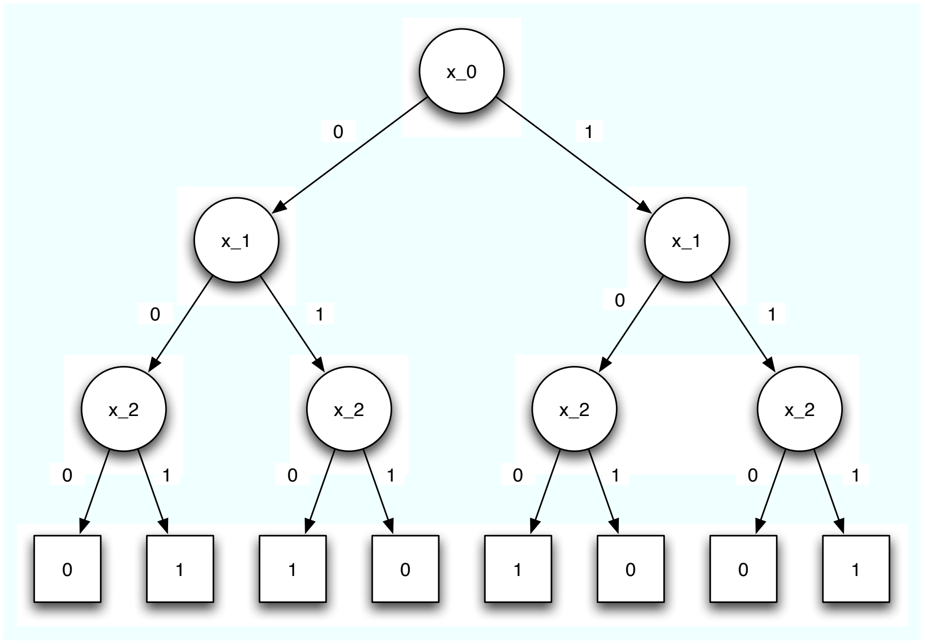

Figure 3.2: Decision tree for the function $f(x_0, x_1, x_2) \doteq x_0 \oplus x_1 \oplus x_2$.

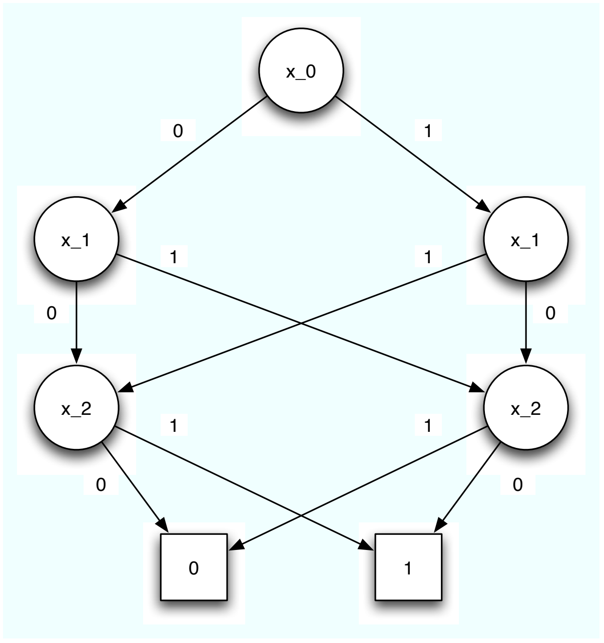

Figure 3.3: Decision diagram for the function $f(x_0, x_1, x_2) \doteq x_0 \oplus x_1 \oplus x_2$.

- Given ROBDD(f), ROBDD($\exists x_i\, f$) and ROBDD($\forall x_i\, f$)[2] can be computed in time $O(|\mathrm{ROBDD}(f)|^2)$.

We noted before that ROBDDs can be a succinct way of representing boolean functions. In fact, the example above on the ternary parity function can be generalized to the n-bit parity function: it can be showed that, while the corresponding binary tree has $2^{n+1} - 1$ nodes, the ROBDD representation has only $2n + 1$ nodes. Hence ROBDDs can be exponentially more succinct. Unfortunately, the size of the ROBDD representation strongly depends on the ordering of variables. In fact, there are cases where ROBDDs still have an exponential size (in the number of variables) for some classes of variable orderings and, worse, finding the optimal variable ordering is infeasible. In fact, even checking that a given variable ordering is optimal is NP-complete. Worst of all, there are boolean functions whose ROBDD is of exponential size for all variable orderings. In a certain sense this should be somewhat obvious, as there are 2^{2^n} boolean functions over n boolean arguments, and it would be surprising that we would be able to represent so many functions with only $O(n)$ memory!

However, in practical cases (especially in hardware systems) boolean formulae have an inherent redundancy and their ROBDD representations have small size. Heuristics[3] are needed for quickly finding near-optimal variable orderings and *dynamic reordering* techniques are used to periodically reduce the size of ROBDDs.

3.3 Using ROBDDs in symbolic model-checking

The advantage of ROBDDs comes from the fact that they can *symbolically represent* arbitrary subsets of the state space $\mathcal{S}$. This makes ROBDDs a useful means in iterative model-checking algorithms. Let us sketch why.

We associate ROBDDs to a Kripke structure $\mathcal{M} = (\mathcal{S}, \mathcal{R}, \mathcal{L})$ with atomic propositions in AP (see 2.2, on page 15). We shall assume that, without loss of generality, $|\mathcal{S}| = 2^n$, for some n. Then, by using n boolean variables $x_1, \ldots, x_n$ we can represent every element of $\mathcal{S}$ by an assignment of 0's and 1's to the x_i's. (This is of no loss of generality since every finite set A can be encoded with at most $\lceil \log_2 |A| \rceil$ boolean variables.) Hence, a boolean formula $f(x_1, \ldots, x_n)$ represent a subset of $\mathcal{S}$, namely the subset represented by all the assignments of 0-1 values to the x_i's that makes f equal to 1. For every variable x_i there exists a corresponding *primed variable* x_i', which intuitively represent the value of x_i at the next step. We will use vector notation and write $\vec{x}$ for $x_1, \ldots, x_n$. We build the following ROBDDs:

[2]Boolean quantifiers $\forall$ and $\exists$ are defined as: $\exists x\, f \doteq f[0/x] \vee f[1/x]$ and $\forall x\, f \doteq f[0/x] \wedge f[1/x]$, where $f[b/x]$ is f with all occurrences of x replaced by b.

[3]This term was introduced by Allen Newell, as denoting a process that *can* solve a problem, but is not guaranteed to succeed. Heuristic methods are opposed to *algorithmic* ones, which always succeed. Newell himself took the term from the work of his professor George Pólya, the author of the celebrated book "How to solve it".

- For each $p \in AP$, $\mathrm{ROBDD}_p(\vec{x})$ is the ROBDD that represents all the states in which p is true, i.e., a variable assignments $\vec{y} \in \{0,1\}^n$ satisfies $\mathrm{ROBDD}_p(\vec{x})$ iff $\vec{y}$ denotes a state in $\mathcal{L}^{-1}(p)$.
- $\mathrm{ROBDD}_{\mathcal{R}}(\vec{x}, \vec{x}')$ is the ROBDD for the transition relation. A variable assignment $(\vec{y}_1, \vec{y}_2) \in \{0,1\}^{2n}$ satisfies $\mathrm{ROBDD}_{\mathcal{R}}(\vec{x}, \vec{x}')$ iff $\vec{y}_1$ denotes a state s_1, $\vec{y}_2$ denotes a state s_2 and
$$(s_1, s_2) \in \mathcal{R} \ .$$

Next, we show how to associate ROBDDs to μ-calculus formulae. Let $\mathcal{A}$ be an environment for relational variables, i.e., $\mathcal{A}$ assigns ROBDDs to relational variables. As usual, with $\mathcal{A}[R \leftarrow B]$ we denote the updated environment, returning the ROBDD B for the relational variable R. The translation from a μ-calculus formula f to the corresponding ROBDD is given by the function $\mathcal{B}_{\mathcal{A}}$, which is defined by induction on the structure of f:

- $\mathcal{B}_{\mathcal{A}}(p) = \mathrm{ROBDD}_p(\vec{x})$.
- $\mathcal{B}_{\mathcal{A}}(Q) = \mathcal{A}(Q)$.
- $\mathcal{B}_{\mathcal{A}}(\neg f) = \neg \mathcal{B}_{\mathcal{A}}(f)$.
- $\mathcal{B}_{\mathcal{A}}(f \wedge g) = \mathcal{B}_{\mathcal{A}}(f) \wedge \mathcal{B}_{\mathcal{A}}(g)$.
- $\mathcal{B}_{\mathcal{A}}(f \vee g) = \mathcal{B}_{\mathcal{A}}(f) \vee \mathcal{B}_{\mathcal{A}}(g)$.
- $\mathcal{B}_{\mathcal{A}}(\mathbf{EX} f) = \exists \vec{x}' \, (\mathrm{ROBDD}_{\mathcal{R}}(\vec{x}, \vec{x}') \wedge \mathcal{B}_{\mathcal{A}}(f)(\vec{x}'))$.
- $\mathcal{B}_{\mathcal{A}}(\mu Q.\ f(Q)) = \mathrm{MUFIX}(f, \mathcal{A})$.
- $\mathcal{B}_{\mathcal{A}}(\nu Q.\ f(Q)) = \mathrm{NUFIX}(f, \mathcal{A})$.

(Note that the boolean operations on the right hand sides are operations on ROBDDs.) The $\mathrm{MUFIX}(\cdot,\cdot)$ and $\mathrm{NUFIX}(\cdot,\cdot)$ operations (see figure 3.4) are the ROBDD version of the simple iterative algorithms shown before in figure 3.1.

3.4 Complexity considerations

In Section 3.1 we presented a basic method for μ-calculus model-checking. More efficient methods will be presented in Chapter 5. However, all known *explicit-state* algorithms are of complexity $n^{O(|\phi|)}$ (in the worst case), where $n = |\mathcal{S}|$ and $|\phi|$ is the length of the formula. One could ask if *all* explicit-state μ-calculus model-checking algorithms necessarily have complexity proportional to $n^{O(|\phi|)}$, i.e., if μ-calculus model-checking is inherently difficult and no polynomial algorithm exists for arbitrary μ-calculus formulae. Well, this is a longstanding open problem and its is cited in several papers [BCJM96, Eme97, EJS01]. Various attempts have been made at solving it. The first relevant result was in [EJS93], where it was proved that the model-checking problem for the μ-calculus is in

Algorithm 3.3: Minimal fixpoint evaluation using ROBDDs.

```
Function MUFIX($f(Q), \mathcal{A}$)
input  : A $\mu$-calculus formula $f$ and an environment $\mathcal{A}$
output: The ROBDD representing the least fixpoint of $f$
1 $Q'_{\text{bdd}} := \text{FALSE}_{\text{bdd}}$
2 repeat
3     $Q_{\text{bdd}} := Q'_{\text{bdd}}$
4     $Q'_{\text{bdd}} := \mathcal{B}_{\mathcal{A}'}(f)$, where $A' := A[Q \leftarrow Q_{\text{bdd}}]$
5 until $Q_{\text{bdd}} \neq Q'_{\text{bdd}}$
6 return $Q_{\text{bdd}}$
```

Algorithm 3.4: Maximal fixpoint evaluation using ROBDDs.

```
Function NUFIX($f(Q), \mathcal{A}$)
input  : A $\mu$-calculus formula $f$ and an environment $\mathcal{A}$
output: The ROBDD representing the greatest fixpoint of $f$
1 $Q'_{\text{bdd}} := \text{TRUE}_{\text{bdd}}$
2 repeat
3     $Q_{\text{bdd}} := Q'_{\text{bdd}}$
4     $Q'_{\text{bdd}} := \mathcal{B}_{\mathcal{A}'}(f)$, where $A' := A[Q \leftarrow Q_{\text{bdd}}]$
5 until $Q_{\text{bdd}} \neq Q'_{\text{bdd}}$
6 return $Q_{\text{bdd}}$
```

Figure 3.4: Simple iterative evaluation of fixpoints with ROBDDs.

NP$\cap$coNP, hence it is generally argued that it is unlikely for the problem to be NP-complete. Moreover, in [CGP00] it is conjectured that no polynomial algorithm does exist. Moreover, it is known from [Bra97] and [Len96] that the μ-calculus hierarchy is strict, i.e., there exist complex formulae with high alternation depth that are not equivalent to any formula with fewer alternations. This implies that actual algorithms cannot be made polynomial by rewriting complex formulae in terms of simpler ones. The complexity of μ-calculus model-checking is probably the most important unresolved theoretical problem about the μ-calculus.

Also the complexity of *symbolic model-checking* has been studied. In general, if we have n separate modules of finite size, when we asynchronously combine them, we obtain a structure of size $|\mathcal{S}| = 2^{O(n)}$. This is what is commonly known as the *state-space explosion* problem. As we have seen, symbolic techniques aim at limiting the state-space explosion, trying to represent the system in $O(n)$ space. Hence, the symbolic model-checking problem is to be formulated with respect to the parameter n, and not $|\mathcal{S}|$. One could ask if symbolic model-checking is always convenient with respect to explicit-state model-checking. In [Rab00] it is proved that this is not the case. There exists a single μ-calculus formula s.t. the symbolic μ-calculus model-checking problem requires *at least* $O(c^{n/\log^2 n})$ time, for some $c > 1$. Hence, in this sense, symbolic model-checking is in general no better than explicit-state model-checking. It is an open problem whether the lower bound can be raised to $O(c^n)$, matching the known upper bound.

3.5 Formal verification with NuSMV

In this section we present a simple example of model-checking. Throughout the thesis, and especially in Chapter 7, we will use the NuSMV symbolic model-checker. NuSMV is developed by the joint effort of CMU (Carnegie Mellon University) and ITC-IRST (Istituto per la Ricerca Scientifica e Tecnologica di Trento). See [CCG$^+$02]. NuSMV is the result of the complete re-engineering, re-implementation and extension of the old CMU SMV model-checker. With version 2.0, NuSMV became an open-source project. NuSMV accept specifications in the **CTL**, **LTL** and (the recent) **PSL** temporal logics.

We will show the functionalities of NuSMV through an example. Consider figure 3.5. A simple mutual exclusion protocol with two processes is implemented. The declaration `MODULE prc` defines the generic process. The behaviour of process 1 is the following:

1. Remain a whatever amount of time in the non-critical region, hence go to the next step.

2. Try to access the critical region: if the other process is in its non-critical region or it is trying to enter, but it is our turn, then enter the critical region (next step). Otherwise, wait a unit of time and repeat.

3. Remain a finite amount of time in the critical region, then reset the turn and go to first step.

The behaviour of process 2 is symmetric and can be deduced from figure 3.5. At each step a NuSMV process is selected for execution, following the established paradigm of modeling concurrency by the interleaving of actions. We need that each process is executed infinitely often, hence the declaration `FAIRNESS running`. Moreover, we must ensure that processes do not remain in their critical regions forever. This is accomplished by the other fairness constraint `FAIRNESS !(state0 = critical)`.

In this example, there are three **CTL** properties:

1. Mutual exclusion: it is not the case that both processes are in their critical section,

   ```
   SPEC
     AG !(s0 = critical & s1 = critical)
   ```

2. No starvation: if the first process wants to enter the critical region, then it will eventually succeed,

   ```
   SPEC
     AG (s0 = trying -> AF s0 = critical)
   ```

3. Bounded overtaking: process one cannot enter its critical region two consecutive times,

   ```
   SPEC
     AG (s0 = critical -> A[s0 = critcal U !(s0 = critical) &
     A[!(s0 = critical) U (s1 = critical)]])
   ```

NuSMV was able to instantaneously check these properties. We will return to other practical examples of NuSMV models in Chapter 6, and to the mutual exclusion problem in Chapter 7.

```
MODULE main

VAR
s0: noncritical, trying, critical;
s1: noncritical, trying, critical;
turn: boolean;

pr0: process prc(s0, s1, turn, 0);
pr1: process prc(s1, s0, turn, 1);

ASSIGN init(turn) := 0;

SPEC AG !(s0 = critical & s1 = critical)

SPEC AG (s0 = trying -> AF s0 = critical)

SPEC AG (s0 = critical & s1 = trying ->
 A[s0 = critical U !(s0 = critical) &
 A[!(s0 = critical) U (s1 = critical)]])

MODULE prc(state0, state1, turn, turn0)

ASSIGN init(state0) := noncritical;

next(state0) := case
   (state0 = noncritical) : trying,noncritical;
   (state0 = trying) & (state1 = noncritical): critical;
   (state0 = trying) & (state1 = trying) &
                               (turn = turn0): critical;
   (state0 = critical) : critical,noncritical;
   1: state0;
esac;

next(turn) := case
   turn = turn0 & state0 = critical: !turn;
   1: turn;
esac;

FAIRNESS running
FAIRNESS !(state0 = critical)
```

Figure 3.5: Simple mutual exclusion algorithm in NuSMV, with fair **CTL** specifications.

Part II

Development: A New Logic and a New Model-Checking Algorithm

Chapter 4

A temporal logic with fairness constraints

In this chapter we introduce the novel temporal logic ω-***CTL***. In Section 4.1 we define the notion of ω-regular languages over paths and in Section 4.2 we introduce the logic ω-**CTL**. In Section 4.3 we show how ω-**CTL** is capable of expressing common kinds of fairness constraints and in Section 4.4 we give an encoding of ω-**CTL** in μ-calculus. Finally, in Section 4.5 we show how (a superset of) **CTL** can be succinctly translated into ω-**CTL**.

4.1 ω-regular languages over paths

In this section we present a formalism for modeling computation paths in a Kripke structure. We interpret sets of (finite and infinite) paths of a Kripke structure $\mathcal{M}$ as ω-regular languages over the states of $\mathcal{M}$. We follow the presentation of [Par81], adapted to our specific needs.

Consider a Kripke structure $\mathcal{M} = (\mathcal{S}, \mathcal{S}_0, \mathcal{R})$, where $\mathcal{S}$ is a non-empty set of states, $\mathcal{S}_0 \subseteq \mathcal{S}$ is the set of initial states and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation, which we assume to be total in the first argument, i.e., for all $s \in \mathcal{S}$ there exists $s' \in \mathcal{S}$ s.t. $(s, s') \in \mathcal{R}$. A *path* π is a finite or infinite sequence of states (not necessarily distinct) satisfying $\mathcal{R}$, that is:

$$\pi = s_0 s_1 \dots \ ,$$

and $(s_i, s_{i+1}) \in \mathcal{R}$, for each i; we define $\pi(i) \doteq s_i$[1]. We write $|\pi| \doteq n$ when π consists of exactly $n \in \mathbb{N}$ states. Otherwise, π is infinite and $|\pi| \doteq \infty$.

Note that the notion of path used here slightly differs from the notion of path presented earlier in Chapter 2. In fact, while on the one hand in Chapter 2 paths are essentially *fullpaths*, i.e., paths of infinite length, on the other hand

[1] A more formal definition would view a path π as a function from some subset of $\mathbb{N}$ to $\mathcal{S}$.

here we have paths described by ω-regular expressions, which can be either finite or infinite. Hence, the reader should be careful of the different definition of path used in this chapter.

With $\mathcal{S}^+$ we denote the set of all (nonempty) finite paths and with $\mathcal{S}^\omega$ we denote the set of all infinite paths; we do not mention neither $\mathcal{M}$ nor $\mathcal{R}$ as it should be clear from the context to which Kripke structure we do refer. The totality of $\mathcal{R}$ implies that both $\mathcal{S}^+$ and $\mathcal{S}^\omega$ are non-empty and, moreover, that $\mathcal{S}^+$ has a infinite number of elements.

With ε we denote a distinguished, special *null* path, satisfying $|\varepsilon| \doteq 0$; we will treat ε separately in what follows. As usual, we define $\mathcal{S}^* \doteq \{\varepsilon\} \cup \mathcal{S}^+$ and $\mathcal{S}^\infty \doteq \mathcal{S}^* \cup \mathcal{S}^\omega$.

Now we define concatenation of paths. Two paths $\pi_1 \in \mathcal{S}^+, \pi_2 \in \mathcal{S}^\infty$ are *joinable* if $|\pi_1| = n$ and $(\pi_1(n-1), \pi_2(0)) \in \mathcal{R}$. For $\pi \in \mathcal{S}^\infty$ and joinable paths $\pi_1 \in \mathcal{S}^+$ and $\pi_2 \in \mathcal{S}^\infty$:

$$\begin{aligned}
\varepsilon\pi &\doteq \pi && (4.1)\\
\pi_1\varepsilon &\doteq \pi_1 && (4.2)\\
\pi_1\pi_2 &\doteq \pi_1(0)\pi_1(1)\ldots\pi_1(n-1)\pi_2(0)\pi_2(1)\ldots && (4.3)
\end{aligned}$$

A *ω-regular language of paths* L over $\mathcal{M}$ is a subset of $\mathcal{S}^\infty$. We can extract the finite and infinite paths of L:

$$\begin{aligned}
\mathcal{F}in(L) &\doteq L \cap \mathcal{S}^* && (4.4)\\
\mathcal{I}nf(L) &\doteq L \cap \mathcal{S}^\omega\ . && (4.5)
\end{aligned}$$

Clearly, $\mathcal{F}in(L)$ and $\mathcal{I}nf(L)$ are a partition of L, i.e., $L = \mathcal{F}in(L) \cup \mathcal{I}nf(L)$ and $\mathcal{F}in(L) \cap \mathcal{I}nf(L) = \emptyset$.

We say that a language L is *regular* when $L = \mathcal{F}in(L)$, i.e., L consists only of finite sequences.

Operations on ω-regular languages

Consider two ω-regular languages $A, B \subseteq \mathcal{S}^\infty$. The *concatenation* of A and B is defined as

$$A \circ B \doteq \{\ \pi_1\pi_2 \mid \pi_1 \in \mathcal{F}in(A), \pi_2 \in B\ \} \cup \mathcal{I}nf(A)\ . \qquad (4.6)$$

Define $A^0 \doteq \{\varepsilon\}$, $A^{i+1} \doteq A \circ A^i$. (From the definitions above, it is easy to verify that $A = A^1$.) Now we can define other operations on languages, namely

$$\begin{aligned}
&\text{(star-closure)} & A^* &\doteq \bigcup_{i\geq 0} A^i && (4.7)\\
&\text{(omega-closure)} & A^\omega &\doteq \mathcal{S}^\infty \text{ if } \varepsilon \in A &&\\
& & &\doteq \{\ \pi_0\pi_1\ldots \mid \pi_i \in A\ \} \cup \mathcal{I}nf(A^*) \text{ else} && (4.8)\\
&\text{(dagger-closure)} & A^\infty &\doteq A^* \cup A^\omega\ . && (4.9)
\end{aligned}$$

ω-regular expressions

With the usual overloading of symbols, we define *ω-regular expressions of paths* as the language generated from the following rules

$$a \ ::= \ f \mid a + a \mid a \cdot a \mid a^* \mid a^\omega \tag{4.10}$$

where f ranges over the syntactic set of state formulae (defined later); for now, it only cares to know that f denotes the set of states $[\![f]\!]$ in which f is true. We will feel free to indicate (syntactic) concatenation also with juxtaposition. Also note that, if from the previous grammar we omit the last rule, we obtain ordinary regular expressions. Usual abbreviations are

$$a^+ \ \doteq \ a \cdot a^* \tag{4.11}$$
$$a^\infty \ \doteq \ a^* + a^\omega \tag{4.12}$$
$$\varepsilon \ \doteq \ \mathbf{false}^* \ . \tag{4.13}$$

(**false** is a state formula holding in no state, i.e., $[\![\mathbf{false}]\!] = \emptyset$.)

We define the denotation of a ω-regular expression a in the straightforward way: $\mathcal{L}\,(a) \subseteq \mathcal{S}^\infty$ is the language defined inductively by

$$\mathcal{L}\,(f) \ \doteq \ \{ \ \pi = s \ \mid s \in [\![f]\!] \ \} \tag{4.14}$$
$$\mathcal{L}\,(a+b) \ \doteq \ \mathcal{L}\,(a) \cup \mathcal{L}\,(b) \tag{4.15}$$
$$\mathcal{L}\,(a \cdot b) \ \doteq \ \mathcal{L}\,(a) \circ \mathcal{L}\,(b) \tag{4.16}$$
$$\mathcal{L}\,(a^*) \ \doteq \ \mathcal{L}\,(a)^* \tag{4.17}$$
$$\mathcal{L}\,(a^\omega) \ \doteq \ \mathcal{L}\,(a)^\omega \ , \tag{4.18}$$

where in the first clause we identify states with paths of length one, and in the remaining clauses the operations on the right-hand sides are the operations on languages introduced earlier.

When two expressions a and b denote the same language, they are *semantically equivalent*, and we write

$$a \equiv b \quad \text{iff} \quad \mathcal{L}\,(a) = \mathcal{L}\,(b) \ . \tag{4.19}$$

Dually to the semantic operations on languages $\mathcal{F}in(\cdot)$ and $\mathcal{I}nf(\cdot)$ introduced earlier, we define the corresponding syntactic operations on expressions: $\mathrm{fin}(a)$ and $\mathrm{inf}(a)$. The definition of $\mathrm{fin}(\cdot)$ is[2]

$$\mathrm{fin}(f) \ \doteq \ f \tag{4.20}$$
$$\mathrm{fin}(a+b) \ \doteq \ \mathrm{fin}(a) + \mathrm{fin}(b) \tag{4.21}$$
$$\mathrm{fin}(a \cdot b) \ \doteq \ \mathrm{fin}(a) \cdot \mathrm{fin}(b) \tag{4.22}$$

[2]We take the liberty of interpreting state formulae as paths of length one. Moreover, with **false** we denote the empty set of paths $\emptyset$, as previously said. Similarly, with **true** we denote $\mathcal{S}$.

$$\begin{aligned} \mathrm{fin}(a^*) &\doteq \mathrm{fin}(a)^* && (4.23)\\ \mathrm{fin}(a^\omega) &\doteq \textbf{true}^* \text{ if } \varepsilon \in \mathcal{L}(a) \\ &\doteq \textbf{false} \text{ otherwise },&& (4.24) \end{aligned}$$

while the definition of $\inf(\cdot)$ is

$$\begin{aligned} \inf(f) &\doteq \textbf{false} && (4.25)\\ \inf(a+b) &\doteq \inf(a)+\inf(b) && (4.26)\\ \inf(a\cdot b) &\doteq \mathrm{fin}(a)\cdot\inf(b)+\inf(a) && (4.27)\\ \inf(a^*) &\doteq \mathrm{fin}(a)^*\cdot\inf(a) && (4.28)\\ \inf(a^\omega) &\doteq \textbf{true}^\omega \text{ if } \varepsilon\in\mathcal{L}(a)\\ &\doteq \inf(a^*)+\mathrm{fin}(a)^\omega \text{ otherwise }. && (4.29) \end{aligned}$$

It is easy to prove that $\mathrm{fin}(a)$ and $\inf(a)$ respect the meaning of expressions, i.e.,

$$\begin{aligned} \mathcal{F}in(\mathcal{L}(a)) &= \mathcal{L}(\mathrm{fin}(a)) \text{ , and}\\ \mathcal{I}nf(\mathcal{L}(a)) &= \mathcal{L}(\inf(a)) \text{ .} \end{aligned}$$

Finally, we say that the expression a is *finitary* when $a \equiv \mathrm{fin}(a)$, and *infinitary* when $a \equiv \inf(a)$.

Normal form, ε-free expressions

ω-regular expressions can be put in normal form [Cho74, Par81]:

Theorem 4.1 (Normal form for ω-regular expressions)**.** *Every ω-regular expression a can be written in the form*

$$a \equiv \mathrm{fin}(a)+\inf(a) \text{ ,} \tag{4.30}$$

where $\inf(a)$ *is in the form of a finite sum of ω-regular expressions:*

$$\inf(a) = \sum_{i=1}^{n} b_i(c_i)^\omega \text{ ,} \tag{4.31}$$

for some $n \in \mathbb{N}$ and where all $\mathcal{L}(b_i), \mathcal{L}(c_i)$ are regular languages and $\varepsilon \notin \mathcal{L}(c_i)$.

This theorem will be useful when we will define the logic ω-**CTL**, allowing us to consider only expressions in normal form.

We say that an expression a is *ε-free* if $\varepsilon \notin \mathcal{L}(a)$. We can check if a is ε-free with the following induction:

$$\begin{aligned} \text{is\_}\varepsilon\text{\_free}(f) &\quad \text{holds} && (4.32)\\ \text{is\_}\varepsilon\text{\_free}(a+b) &\iff \text{is\_}\varepsilon\text{\_free}(a) \text{ and is\_}\varepsilon\text{\_free}(b) && (4.33)\\ \text{is\_}\varepsilon\text{\_free}(a\cdot b) &\iff \text{is\_}\varepsilon\text{\_free}(a) \text{ or is\_}\varepsilon\text{\_free}(b) && (4.34)\\ \text{is\_}\varepsilon\text{\_free}(a^\omega) &\iff \text{is\_}\varepsilon\text{\_free}(a) \text{ .} && (4.35) \end{aligned}$$

Note that we deliberately omitted the clause for a^*, since a^* is clearly not ε-free.

We can take a step further and transform a into the "closest" expression b which is ε-free, i.e., $\mathcal{L}(b) = \mathcal{L}(a) \setminus \{\varepsilon\}$. $b \doteq \text{mk\_}\varepsilon\text{\_free}(a)$ is given by the following structural induction:

$$\begin{aligned}
\text{mk\_}\varepsilon\text{\_free}(f) &\doteq f && (4.36)\\
\text{mk\_}\varepsilon\text{\_free}(a+b) &\doteq \text{mk\_}\varepsilon\text{\_free}(a) + \text{mk\_}\varepsilon\text{\_free}(b) && (4.37)\\
\text{mk\_}\varepsilon\text{\_free}(a \cdot b) &\doteq a \cdot b \text{ if is\_}\varepsilon\text{\_free}(a) &&\\
&\doteq \text{mk\_}\varepsilon\text{\_free}(b) + \text{mk\_}\varepsilon\text{\_free}(a) \cdot b \text{ else} && (4.38)\\
\text{mk\_}\varepsilon\text{\_free}(a^*) &\doteq \text{mk\_}\varepsilon\text{\_free}(a)^+ && (4.39)\\
\text{mk\_}\varepsilon\text{\_free}(a^\omega) &\doteq \text{mk\_}\varepsilon\text{\_free}(a)^\omega \ . && (4.40)
\end{aligned}$$

Finally, for each expression a we associate its *ω-regular extension* $\omega\text{-ext}(a)$, defined as follows:

$$\omega\text{-ext}(a) \doteq \begin{cases} a \cdot (\mathbf{true})^\omega & \text{if } a \equiv \text{fin}(a) \\ a & \text{otherwise .} \end{cases} \quad (4.41)$$

4.1.1 The significance of the existence of a normal form

This is a short digression about the role of the existence of a normal form in ω-regular expressions. It is shown in [Par81] that ω-regular expressions are equivalent to the *linear-time* μ-calculus, i.e., the version of μ-calculus that comes out when we interpret its formulae on linear sequences rather than on trees. In this sense, we can see ω-regular expressions as a linear-time formalism. The correspondence of ω-regular expressions and the linear μ-calculus can be intuitively justified by noting that 1) star-closure is like a minimal fixpoint, and 2) omega-closure is like a maximal fixpoint. That ω-regular expressions can be put in normal form is known from [Cho74] (already mentioned). Combining this with [Par81], we deduce that also linear μ-calculus expressions have normal forms. This implies that arbitrarily complex linear μ-calculus formulae can always be translated into equivalent formulae of alternation depth (at most) 2. Hence, in linear-time μ-calculus the hierarchy "collapses" to depth 2. On the contrary, in branching-time μ-calculus [Bra97] and [Len96] showed that, as the alternation depth grows, we get a true hierarchy of μ-calculus formulae of increasing expressive power.

4.2 A temporal logic over ω-regular expressions

We recall here the definition of Kripke structure:

Definition 4.1 (Kripke structure)**.** *Let AP be a set of atomic propositions. A* Kripke structure *$\mathcal{M}$ over AP is a 4-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{L})$, where*

- *$\mathcal{S}$ is a non-empty finite set of states,*

- $\mathcal{S}_0 \subseteq \mathcal{S}$ *is the set of initial states,*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ *is a transition relation over* $\mathcal{S}$*. It is total, i.e., for all* $s \in \mathcal{S}$ *there exists* $s' \in \mathcal{S}$ *s.t.* $(s, s') \in \mathcal{R}$*.*
- $\mathcal{L} : \mathcal{S} \to \mathcal{P}(AP)$ *is a map that labels each state in* $\mathcal{S}$ *with the atomic propositions holding in that state.*

We need a total transition relation as we are interested in infinite, non-terminating computations. This is not a limitation, since we can extend every transition relation $\mathcal{R}$ into a total transition relation $\mathcal{R}'$ by adding loops over terminal states, i.e., $\mathcal{R}' = \mathcal{R} \cup \{\ (s, s)\ \mid \forall s' \in \mathcal{S}\ (s, s') \notin \mathcal{R}\ \}$.

We model computations over $\mathcal{M}$ as ω-regular expressions a over $\mathcal{M}$. We say that π is an *a-path* if π is a sequence of states in $\mathcal{L}(a)$. As we are interested in non-terminating computations, we will give meaning only to expressions denoting infinite paths, that is expressions a s.t. $a \equiv \inf(a)$ (or, equivalently, $\mathcal{L}(a) = \mathcal{L}(a) \cap \mathcal{S}^\omega$).

We are now ready to introducte our logic ω-**CTL**. In ω-**CTL** it is possible specify two kind of properties, following the tradition started in [Lam77]: liveness properties and safety properties. Moreover, we relativize properties to a set of paths, introducing two flavours of satisfaction: universal (i.e., for each path in the given set) and existential (i.e., for some path in the given set). We will use the standard **CTL** operators, but the semantics of path quantifiers is modified to reflect the above intuition. For example (for a a ω-regular expression):

- $\mathbf{EG}(a, f)$ means that there exists a path π described by a s.t. f holds in each state of π.
- $\mathbf{AF}(a, f)$ means that f eventually holds in each path described by a.

In ω-**CTL** there are only state formulae f. The syntax is:

$$f \ ::= \ p \mid f \ \vee \ f \mid \neg f \mid \mathbf{EG}(a, f) \mid \mathbf{EU}(a, f, f) \tag{4.42}$$

where p ranges over AP and a is a ω-regular expression.

Other operators are introduced via the definitions[3]:

$$\begin{aligned}
f \wedge g &\doteq (\neg f) \vee (\neg g) && (4.43)\\
\mathbf{EF}(a, f) &\doteq \mathbf{EU}(a, \mathbf{true}, f) && (4.44)\\
\mathbf{AG}(a, f) &\doteq \neg \mathbf{EF}(a, \neg f) && (4.45)\\
\mathbf{AF}(a, f) &\doteq \neg \mathbf{EG}(a, \neg f) && (4.46)\\
\mathbf{AU}(a, f, g) &\doteq \neg \mathbf{EU}(a, \neg g, \neg f \wedge \neg g) \wedge \neg \mathbf{EG}(a, \neg g)\ . && (4.47)
\end{aligned}$$

[3]Even more **CTL** operators can be introduced via similar definitions, see 2.3, on page 18.

When a formula f is true in state s we write $s \models f$. The semantics of ω-**CTL** formulae is:

$$
\begin{array}{lll}
s \models p & \iff & s \in \mathcal{L}(p) \\
s \models f \vee g & \iff & s \models f \text{ or } s \models g \\
s \models \neg f & \iff & \text{not } s \models f \\
s \models \mathbf{EG}(a, f) & \iff & \text{for some } \pi \in \mathcal{L}(a) \text{ and for all } i \leq |\pi|,\ \pi(i) \models f \\
s \models \mathbf{EU}(a, f, g) & \iff & \text{for some } \pi \in \mathcal{L}(a) \text{ and for some } i \leq |\pi|,\ \pi(i) \models g \\
 & & \text{and for all } j < i,\ \pi(j) \models f\ .
\end{array}
$$

(Alternatively, we could view the semantics of f as the set of states in which f holds, i.e., $[\![f]\!] = \{\ s \in \mathcal{S}\ \mid s \models f\ \}$, and give an analogous structural definition.)

4.3 Expressing fairness constraints

We introduce some syntactic sugaring for expressing the notion of *infinitely often* into ω-**CTL** (abbreviated as *i.o.*). For state formulae $f_1, \ldots, f_n$, we define the ω-regular expression:

$$\{f_1, \ldots, f_n\}^{\text{i.o.}} \doteq ((\neg f_1)^* f_1 \ldots (\neg f_n)^* f_n)^\omega\ .$$

It is easy to see that

$$\pi \in \mathcal{L}\left(\{f_1, \ldots, f_n\}^{\text{i.o.}}\right) \iff \text{each of } f_i \text{ occurs infinitely often in } \pi\ .$$

Among the various notions of fairness proposed, we consider here *impartiality* and *weak fairness* (see 2.3, on page 19). For state formulae $a_1, \ldots, a_n$, $b_1, \ldots, b_n$, f we have:

- *Impartiality*: f globally holds in some path with infinitely many a_i's,

$$\mathbf{EG}(\{a_1, \ldots, a_n\}^{\text{i.o.}}, f)\ .$$

- *Weak fairness*: f globally holds in some path with infinitely many $\neg a_i$'s or b_i's,

$$\mathbf{EG}(\{a_1 \Rightarrow b_1, \ldots, a_n \Rightarrow b_n\}^{\text{i.o.}}, f)\ .$$

Note that in ω-**CTL** we cannot (succinctly) express *strong fairness.*

In Chapter 7 we will present more examples of formulae written in ω-**CTL**.

4.4 Translation into μ-calculus

We now show how to translate the logic ω-**CTL** in the propositional μ-calculus of [Koz83]. The translation of $\mathbf{EG}(a, f)$ was inspired from [EJS01], while the translation of $\mathbf{EU}(a, f, g)$ is ours.

We consider only ε-free infinitary ω-regular expressions in *normal form* (see 4.1, on page 42). We give a function $\mathcal{T}r\,(\cdot)$ which maps a formula of ω-**CTL** into a formula of the μ-calculus. We will not prove here the correctness of the translation. Basic cases are straightforward:

$$\mathcal{T}r\,(p) \doteq p \tag{4.48}$$

$$\mathcal{T}r\,(f \vee g) \doteq \mathcal{T}r\,(f) \vee \mathcal{T}r\,(g) \tag{4.49}$$

$$\mathcal{T}r\,(\neg f) \doteq \neg\mathcal{T}r\,(f)\ . \tag{4.50}$$

The translation of the modalities $\mathbf{EG}(\cdot,\cdot)$ and $\mathbf{EU}(\cdot,\cdot,\cdot)$ is less obvious. We need the translations $\mathcal{T}rEG$ and $\mathcal{T}rEU$ for each one of the two modalities. Then we define

$$\mathcal{T}r\,(\mathbf{EG}(a,f)) \doteq \mathcal{T}rEG\,(a,f,\cdot) \tag{4.51}$$

$$\mathcal{T}r\,(\mathbf{EU}(a,f,g)) \doteq \mathcal{T}rEU\,(a,f,g,\cdot)\ . \tag{4.52}$$

(Note that the last argument of both $\mathcal{T}rEG$ and $\mathcal{T}rEU$ has been left unspecified. We will later return on this.)

The function $\mathcal{T}rEG$

Consider the modality $\mathbf{EG}(a,f)$. We define $g' \doteq \mathcal{T}rEG\,(a,f,x)$ with respect to a *given* μ-calculus formula x. We have that g' is a μ-calculus formula which holds in a state s iff there exists a path π in $\mathcal{L}\,(a)$ starting at s, f holds in each state of π *and* π is joinable with a state satisfying x[4]. The definition of $\mathcal{T}rEG$ is by structural induction on the expression a and is shown in figure 4.1. Here is an informal explanation of the rules of the above figure.

- (4.53): This case is straightforward. Here f is interpreted as a path of length one, f' holds in the initial (and unique) state and there is a next state where x holds.
- (4.54): By the distributivity of $\mathbf{E}$ on $\vee$, we have that either 1) f always holds on a or 2) f always holds on b, or both.
- (4.55): f always holds on both a and b, and (the path described by) b begins at the end of (the path described by) a.
- (4.56): From the definition of minimal fixpoint, we have that there exists an $n \geq 0$ s.t. f always holds on n (joint) copies of a.
- (4.57): f always holds on an infinite path built from (infinitely many) copies of a.

These definitions are sufficient as we are considering only expressions in normal form. Note that in the last case we do not make use of the third argument of $\mathcal{T}rEG$. This is consistent with the definition of Equation 4.51.

[4]Note that x is once again interpreted as a set of paths of length one.

$$\begin{aligned}
\mathcal{T}rEG\,(f, f', x) &\doteq f \wedge f' \wedge \mathbf{EX}x && (4.53)\\
\mathcal{T}rEG\,(a+b, f, x) &\doteq \mathcal{T}rEG\,(a, f, x) \vee \mathcal{T}rEG\,(b, f, x) && (4.54)\\
\mathcal{T}rEG\,(a \cdot b, f, x) &\doteq \mathcal{T}rEG\,(a, f, \mathcal{T}rEG\,(b, f, x)) && (4.55)\\
\mathcal{T}rEG\,(a^*, f, x) &\doteq \mu\, y.\, x \vee \mathcal{T}rEG\,(a, f, y) && (4.56)\\
\mathcal{T}rEG\,(a^\omega, f, \cdot) &\doteq \nu\, x.\, \mathcal{T}rEG\,(a, f, x) && (4.57)
\end{aligned}$$

Figure 4.1: The translation function $\mathcal{T}rEG$.

The function $\mathcal{T}rEU$

Now consider the modality $\mathbf{EU}(a, f, g)$. Since g must eventually hold in some a-path $\pi \in \mathcal{L}\,(a)$, we will consider only non-null paths π, $|\pi| > 0$, i.e., only those paths π denoted by ε-free expressions. For a ε-free expression a and given a μ-calculus formula x, $\mathcal{T}rEU\,(a, f, g, x)$ is a μ-calculus formula meaning that g eventually holds in some a-path of length > 0, f holds at every previous state of this path and x is joinable with a. The definition of $\mathcal{T}rEU$ is by structural induction on the expression a and is shown in figure 4.2. Here is an informal explanation of the meaning of the those rules:

- (4.58, 4.59): These are identical to equations 4.53 and 4.54, respectively.
- (4.60): We have to consider two cases: either 1) g eventually holds on a or 2) g eventually holds on b. So,
  1. In the first case, we (inductively) ensure that g eventually holds on a, f is true at every preceding state and there is a whatever continuation by b.
  2. In the second case, f always holds in a and there is a continuation by b in which f always holds until g does.

  We require that b is made ε-free in point 2), since if b were to contain the null path, the whole formula could possibly be vacuously true (in particular if b contains $*$'s).
- (4.61): We distinguish two cases (according to the minimal fixpoint semantics):
  1. Basis case. We have that g eventually holds on a, f holds at every preceding state and there is a continuation by 0 or more a's.
  2. Repeating case. f always holds on a and there is a continuation in which we restart again. Note that we cannot restart again forever (minimal fixpoint), hence we must eventually reach the base case only after a finite occurrences of the repeating case.

$$\begin{aligned}
\mathcal{T}r\mathit{EU}\,(f, f', g, x) &\doteq f \wedge g \wedge \mathbf{EX}x && (4.58)\\
\mathcal{T}r\mathit{EU}\,(a+b, f, g, x) &\doteq \mathcal{T}r\mathit{EU}\,(a, f, g, x) \vee \mathcal{T}r\mathit{EU}\,(b, f, g, x) && (4.59)\\
\mathcal{T}r\mathit{EU}\,(a \cdot b, f, g, x) &\doteq \begin{array}{l}\mathcal{T}r\mathit{EU}\,(\text{mk\_}\varepsilon\text{\_free}(a), f, g, \mathcal{T}r\mathit{EG}\,(b, \mathbf{true}, x))\\ \vee\ \mathcal{T}r\mathit{EG}\,(a, f, \mathcal{T}r\mathit{EU}\,(\text{mk\_}\varepsilon\text{\_free}(b), f, g, x))\end{array} && (4.60)\\
\mathcal{T}r\mathit{EU}\left(a^+, f, g, x\right) &\doteq \begin{array}{l}\mu\, y.\ \mathcal{T}r\mathit{EU}\,(a, f, g, \mathcal{T}r\mathit{EG}\,(a^*, \mathbf{true}, x))\ \vee\\ \quad \mathcal{T}r\mathit{EG}\,(a, f, y)\end{array} && (4.61)\\
\mathcal{T}r\mathit{EU}\,(a^\omega, f, g, \cdot) &\doteq \begin{array}{l}\mu\, y.\ \mathcal{T}r\mathit{EU}\,(a, f, g, \mathcal{T}r\mathit{EG}\,(a^\omega, \mathbf{true}, \cdot)) \vee\\ \quad \mathcal{T}r\mathit{EG}\,(a, f, y)\end{array} && (4.62)
\end{aligned}$$

Figure 4.2: The translation function $\mathcal{T}r\mathit{EU}$.

- (4.62): This is case is similar to the previous. The only difference is in the base case, where instead of a continuation by a^*, we have a continuation by a^ω.

It is easy to check that the translation of figure 4.2 is well-defined, that is the function $\mathcal{T}r\mathit{EU}\,(a, \cdot, \cdot, \cdot)$ is only applied to ε-free a's. Moreover, note that the clause for a^* is absent, as clearly a^* is not ε-free, and it has been substituted by a^+ (which is ε-free when a is so).

Finally, note that, when possible, in the definition of $\mathcal{T}r\mathit{EU}$ we make use of $\mathcal{T}r\mathit{EG}$, which turns out to result into simpler μ-calculus formulae.

An example

For an example of application of the above procedure, consider the translation into μ-calculus of the formula[5]

$$\mathbf{AG}\ \left(\texttt{send} \Rightarrow \mathbf{AF}(\{\texttt{running}\}^{\text{i.o.}}, \texttt{receive})\right)\ ,$$

which, for instance, can be used to model the fact that, under fair scheduling, whenever a message has been sent, it is eventually received. To simplify the following expressions, we define $a \doteq \texttt{send}$, $b \doteq \texttt{running}$ and $c \doteq \texttt{receive}$. The translation is accomplished with the following steps:

$$\begin{aligned}
\mathcal{T}r\left(\mathbf{AG}\ (a \Rightarrow \mathbf{AF}(\{b\}^{\text{i.o.}}, c))\right) &= \nu\, X_1.\ \mathcal{T}r\left(a \Rightarrow \mathbf{AF}(\{b\}^{\text{i.o.}}, c)\right) \wedge \mathbf{AX}X_1 =\\
&= \nu\, X_1.\ (a \Rightarrow f) \wedge \mathbf{AX}X_1,\ \text{where } f \text{ is}
\end{aligned}$$

$$\begin{aligned}
f &= \mathcal{T}r\left(\mathbf{AF}(\{b\}^{\text{i.o.}}, c)\right) =\\
&= \neg\mathcal{T}r\left(\mathbf{EG}(\{b\}^{\text{i.o.}}, \neg c)\right) =
\end{aligned}$$

[5]Note that standard **CTL** operators like **AG** are translated using the usual fixpoint representation given at the end of Chapter 2.

$$\begin{aligned}
&= \neg \mathcal{T}rEG\,(((\neg b)^* \cdot b)^\omega, \neg c, \cdot) = \\
&= \neg \nu\, X_2.\ \mathcal{T}rEG\,((\neg b)^* \cdot b, \neg c, X_2) = \\
&= \neg \nu\, X_2.\ \mathcal{T}rEG\,((\neg b)^*, \neg c, \mathcal{T}rEG\,(b, \neg c, X_2)) = \\
&= \neg \nu\, X_2.\ \mu\, X_3.\ \mathcal{T}rEG\,(b, \neg c, X_2) \vee \mathcal{T}rEG\,(\neg b, \neg c, X_3) = \\
&= \neg \nu\, X_2.\ \mu\, X_3.\ (b \wedge \neg c \wedge \mathbf{EX} X_2) \vee (\neg b \wedge \neg c \wedge \mathbf{EX} X_3) = \\
&= \mu\, X_2.\ \nu\, X_3.\ \neg c \Rightarrow ((b \wedge \mathbf{AX} X_2) \vee (\neg b \wedge \mathbf{AX} X_3))\ ,
\end{aligned}$$

which captures the intended meaning.

4.4.1 Complexity issues

The two translations $\mathcal{T}rEG$ and $\mathcal{T}rEU$ have rules, e.g., the rule for '+', that might involve an exponential blow-up in the size of the resulting μ-calculus formula. This is indeed true for both $\mathcal{T}rEG$ and $\mathcal{T}rEU$, although in a concrete implementation this can be avoided in some extent by sharing common subformulae. For example, in the rule

$$\mathcal{T}rEG\,(a + b, f, x) \doteq \mathcal{T}rEG\,(a, f, x) \vee \mathcal{T}rEG\,(b, f, x)\ ,$$

the occurrences of f and x on the left hand side can be stored once and then referenced by pointers. With analogous considerations, we can see that from a practical point of view $\mathcal{T}rEG$ does not introduce any blow-up, while $\mathcal{T}rEU$ might.

The μ-calculus formula which results from the application of $\mathcal{T}r\,(\cdot)$ has alternation depth exactly equal to 2. This happens since we are considering *infinitary* ω-regular expressions in *normal form*. In fact when a is in normal form, it contains no nested ω's. This implies that the resulting μ-calculus formula f is of the form $\nu x\ \mu y\ \mu z\ \ldots$, i.e., f has a leading maximal fixpoint operator νx and then only minimal fixpoints inside νx. This means that efficient model-checking algorithms can be applied to f.

4.5 Relation to other temporal logics

The logic ω-**CTL** can encode **CTL**, and there are a couple of ways of giving such translation. We will concentrate on the basic modalities $\mathbf{EX}, \mathbf{EG}, \mathbf{EU}$, which are sufficient to encode all **CTL**.

First translation

The translation that we present here makes little use of ω-regular expressions. Given a **CTL** formula f, we apply the translation function $\{f\}^t$ to obtain a

ω-**CTL** formula. The definition of $\{f\}^t$ is by induction on f:

$$\begin{array}{lll} \text{case} & f_1 \vee f_2 & \{f\}^t \doteq \{f_1\}^t \vee \{f_2\}^t \\ \text{case} & \neg f_1 & \{f\}^t \doteq \neg \{f_1\}^t \\ \text{case} & \mathbf{EX} f_1 & \{f\}^t \doteq \mathbf{EX}\{f_1\}^t \\ \text{case} & \mathbf{EG} f_1 & \{f\}^t \doteq \mathbf{EG}(\mathbf{true}^\omega, \{f_1\}^t) \\ \text{case} & \mathbf{E}(f_1\,\mathbf{U}\,f_2) & \{f\}^t \doteq \mathbf{EU}(\mathbf{true}^+, \{f_1\}^t, \{f_2\}^t)\ . \end{array}$$

Note again that we make no specific use of ω-regular expressions, giving a direct mapping from **CTL** modalities to ω-**CTL**'s.

Second translation

The other translation that we give is capable of translating a richer (than **CTL**) fragment of **CTL***, which we call **rCTL***. **rCTL*** is like **CTL***, but path formulae cannot be built from negation or intersection and the only path quantifier is **E**. Formally, **rCTL*** formulae are built from the following rules (where we use p for atomic propositions, f for state formulae and g for path formulae):

- If p is an atomic proposition, then it is also a state formula.
- If f, f_1 and f_2 are state formulae, then so are $\neg f$, $f_1 \wedge f_2$, $f_1 \vee f_2$.
- If g is a path formula, then $\mathbf{E}f$ is a state formula.
- If f is a state formula, then it is also a path formula.
- If g, g_1 and g_2 are path formulae, then so are $g_1 \vee g_2$, $\mathbf{X}\,g$, $\mathbf{F}\,g$, $\mathbf{G}\,g$, $g_1\,\mathbf{U}\,g_2$, $g_1\,\mathbf{W}\,g_2$.

The semantics of **rCTL*** is directly brought from **CTL***. We have that **rCTL*** is strictly less expressive than **CTL***, but it is more expressive than **CTL** and expressively incomparable to **LTL**. Note that not all **CTL** formulae are legal **rCTL*** formulae, but every **CTL** formula can be easily rewritten applying De Morgan's law in order to eliminate $\wedge$'s and **A**'s. Hence, for each **CTL** formula there is an equivalent **rCTL*** formula.

For the purpose of translating **rCTL*** into ω-**CTL**, we will consider only the linear time modalities **X**, **G** and **U**. Then, **F** and **W** can be defined as usual:

$$\mathbf{F}\,g \doteq \mathbf{true}\,\mathbf{U}\,g \tag{4.63}$$

$$g_1\,\mathbf{W}\,g_2 \doteq (g_1\,\mathbf{U}\,g_2) \vee \mathbf{G}\,g_1\ . \tag{4.64}$$

Hence, the restricted syntax of **rCTL*** which we refer to is (for p an atomic proposition, f a state formula and g a path formula)

$$f \doteq p \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid \neg f \mid \mathbf{E}g \tag{4.65}$$

$$g \doteq f \mid g_1 \vee g_2 \mid \mathbf{X}\,g \mid Gg \mid g_1\,\mathbf{U}\,g_2\ . \tag{4.66}$$

The translation of a **rCTL*** state formula f into ω-**CTL** is given by $\{f\}^t$. However, we overload the translation function $\{\cdot\}^t$ in order to apply $\{\cdot\}^t$ to both state and path formulae. For a state formula f, the resulting ω-**CTL** formula $\{f\}^t$ is defined by structural induction on f:

$$\{p\}^t \doteq p \tag{4.67}$$
$$\{f_1 \vee f_2\}^t \doteq \{f_1\}^t \vee \{f_2\}^t \tag{4.68}$$
$$\{f_1 \wedge f_2\}^t \doteq \{f_1\}^t \wedge \{f_2\}^t \tag{4.69}$$
$$\{\neg f\}^t \doteq \neg \{f\}^t \tag{4.70}$$
$$\{\mathbf{E}g\}^t \doteq \mathbf{EG}(\{g\}^t, \mathbf{true})\ . \tag{4.71}$$

For a path formula g which is not a state formula, $\{g\}^t$ is an ω-regular expression, which is defined by induction on g:

$$\{g_1 \vee g_2\}^t \doteq \{g_1\}^t + \{g_2\}^t \tag{4.72}$$
$$\{\mathbf{X}\, g\}^t \doteq \mathbf{true} \cdot \{g\}^t \tag{4.73}$$
$$\{\mathbf{G}\, g\}^t \doteq (\{g\}^t)^\omega \tag{4.74}$$
$$\{g_1 \,\mathbf{U}\, g_2\}^t \doteq (\{g_1\}^t)^\star \cdot \{g_2\}^t\ . \tag{4.75}$$

If we were to compose this translation with the one from ω-**CTL** to μ-calculus (given in the previous section), we would obtain an alternative translation Tr' from **rCTL*** to μ-calculus. We note that the resulting μ-calculus formulae would be all of alternation depth ≤ 2. In particular, if the starting formula is from **CTL**, then the resulting μ-calculus formula would have alternation depth 1. Hence, this translation performs on **CTL** as well as the direct (and simpler) method shown in Chapter 2.

Chapter 5

μ-calculus model-checking

In this chapter we present a novel model-checking algorithm for the μ-calculus. We develop on the naive algorithm presented in Section 3.1. In Section 5.1 we show how simple observations can speed up the naive algorithm, while in Section 5.2 we use monotonicity considerations to efficiently evaluate nested fixpoints of the same type. In Section 5.3 further use of monotonicity allows for an improved algorithm. Finally, in Section 5.4 we gather the main ideas together and we develop our algorithm.

5.1 Basic improvements on the naive algorithm

We have previously described a simple algorithm for evaluating fixpoint expressions (see 3.1, on page 26), which resulted from Kleene's theorem applied to continuous functions. The algorithm has a running time proportional to $O(n^{k+1})$, where $n = |\mathcal{S}|$ is the size of the state space and k is the nesting-depth of fixpoint formulae. Refer to figure 3.1. Intuitively, k counts the level of nesting of least and greatest fixpoints.

For instance, consider the evaluation of the expressions of the form

$$\nu X.\, \psi_1(X, \mu Y.\, \psi_2(X, Y)) \ .$$

For each outer iteration of X we need to reach convergence on Y; since the convergence of X can take up to n steps, we need total number of n^2 steps in the worst case to evaluate the innermost formula. So, the total number of steps is $n^2 + n$, which is near to $O(n^3)$. In general, if we have k nested fixpoints, the total number of steps is

$$\sum_{i=1}^{k} n^i = O(n^{k+1}) \ . \tag{5.1}$$

Various basic considerations can reduce the complexity of the above algorithm. We will explain them by examples:

- Consider the formula

$$\nu X.\, \psi_1(X, \mu Y.\, \psi_2(Y)) \ ,$$

where ψ_2 does not depend on X. Although Y is syntactically nested inside X, actually its evaluation does not depend on the value of X. So, we can evaluate Y once and for all, obtaining a worst case complexity of $n + n$ steps for the whole formula.

- Obviously, repeated occurrences of the same subformula may be evaluated once, but this is rather an implementation than an algorithmic issue. Similarly, in the following formula

$$\nu X.\, \psi \ ,$$

where ψ *does not depend on* X, the evaluation can be done in constant time (with respect to X).

- Finally, consider the formula

$$\mu X.\, \psi_1(X, \mu Y.\, \psi_2(X, Y)) \ ,$$

where alternating fixpoints are of the same type. Also in this case we can do better; the basic observation was made by Emerson and Lei in [EL86] and resulted in the first major improvement in μ-calculus model-checking. Since its importance, we will describe it apart.

5.2 Monotonicity considerations. Emerson and Lei's algorithm

First we need some notation (see [BCJ$^+$94]). We will consider fixpoint variables $X_1, \ldots, X_k$, where X_i is nested in X_j for $j < i$; so X_1 is the outermost variable and X_k the innermost one. We write $X_j^{i_1 \cdots i_j}$ for the value of the i_j-th approximation for X_j after the computation of all i_l-th approximations for the enclosing variables X_l, $1 \leq l \leq j$. When X_j converges, we write $i_j \doteq \omega$. For example X_1^3 is the third approximation for X_1 and $X_2^{2\omega}$ is the fixpoint value of X_2 relative to the second approximation of X_1.

Now, returning to the last example (up to a renomination of variables)

$$\begin{array}{rcl} \psi & \equiv & \mu X_1.\, \psi_1(X_1, \tau(X_1)) \\ \tau(X_1) & \doteq & \mu X_2.\, \psi_2(X_1, X_2) \ , \end{array}$$

we now consider the evaluation of ψ according to the improved algorithm of Emerson and Lei [EL86]. Initially X_1 holds the value $\emptyset$, so $X_1^0 \doteq \emptyset$. Next we compute $\tau(X_1^0)$, proceeding as follows: $X_2^{00} \doteq \emptyset$ and we apply $X_2^{0(i+1)} \doteq \psi_2(X_1^0, X_2^{0i})$ until convergence is achieved, that is $X_2^{0\omega} = \psi_2(X_1^0, X_2^{0\omega})$, which is exactly the required $\tau(X_1^0)$. Now we increase X_1 to $X_1^1 = \psi_1(X_1^0, X_2^{0\omega})$ and we proceed to compute $\tau(X_1^1)$. In the naive algorithm, this would be done as before, by resetting X_2^{10} to $\emptyset$. But, since τ is a monotone predicate transformer (see 2.1, on page 22) and $X_1^0 \subseteq X_1^1$, we have that $\tau(X_1^0) \subseteq \tau(X_1^1)$, i.e., the required fixpoint value $\tau(X_1^1)$ is above the last computed one $\tau(X_1^0)$, and we can avoid resetting X_2 to $\emptyset$, letting $X_2^{10} \doteq X_2^{0\omega}$. The correctness of this operation is stated in the following

Theorem 5.1. *Let τ be a monotone predicate transformer on the complete lattice $\langle S, \subseteq \rangle$. If τ is $\cup$-continuous, then we have that*

1. $\mu\tau = \bigcup_{i\geq 0} \tau^i(\emptyset)$*, and*
2. *if* $A \subseteq \mu\tau$*, then* $\mu\tau = \bigcup_{i\geq 0} \tau^i(A)$*.*

Proof. The first part is the classic Kleene's theorem (see 3.1, on page 26). We now prove the second point. Assume $A \subseteq \mu\tau$. Then $\emptyset \subseteq A \subseteq \mu\tau$ and, by monotonicity, $\tau(\emptyset) \subseteq \tau(A) \subseteq \tau(\mu\tau) = \mu\tau$. Iterating, we have that, for all i, $\tau^i(\emptyset) \subseteq \tau^i(A) \subseteq \mu\tau$ and, taking lubs, $\bigcup_{i\geq 0} \tau^i(\emptyset) \subseteq \bigcup_{i\geq 0} \tau^i(A) \subseteq \mu\tau$, from which we conclude $\bigcup_{i\geq 0} \tau^i(A) = \mu\tau$. □

The essence of this theorem is that to compute a least fixpoint, it suffices to start iterating from a value known to be below the fixpoint itself. In general, $X_2^{i\omega} \subseteq X_2^{(i+1)\omega}$ and we can avoid resetting X_2, letting $X_2^{(i+1)0}$ be $X_2^{i\omega}$. So we continue the computation, eventually reaching the final fixpoint values X_1^ω and $X_2^{\omega\omega}$. Now, we see that the inner computation of X_2 describes an increasing chain

$$X_2^{00} \subseteq X_2^{01} \subseteq \ldots \subseteq X_2^{0\omega} = X_2^{10} \subseteq \ldots \subseteq X_2^{1\omega} = X_2^{20} \subseteq \ldots \subseteq X_2^{\omega\omega} . \tag{5.2}$$

But, since the cardinality of the state space is n, every chain can have at most n increases. We conclude that the total number of inner iterations along the entire computation is only $O(n)$—e.g., instead of $O(n^2)$ when we reset values.

This reasoning can be extended to k nested fixpoints of the same type, obtaining a total number of steps $O(kn)$ (compare it with $O(n^k)$ steps of the simple algorithm). Obviously, all the previous considerations dualize to greatest fixpoints. For the details of the algorithm see [EL86].

All the simplifications discussed until here can be encoded in the definition of alternation-depth d; we will not give formal details here, the reader is referred to [CKS92]. Intuitively, the alternation-depth counts only how many times least and greatest fixpoints alternate: with this definition, we can evaluate fixpoint formulae in roughly $O(n^{d+1})$ steps, since we reset the fixpoint only when an alternation occurs. The worst-case space complexity for storing intermediate values is proportional to k, since each variable X_i must remember its previous value.

5.3 Further monotonicity considerations. The algorithm of Browne et al.

As shown in [BCJ$^+$94], if we store even more intermediate values, we can improve the time bound at the cost of an increase in space complexity. Consider the formula:

$$\psi \equiv \mu X_1.\, \psi_1(X_1, \nu X_2.\, \psi_2(X_1, X_2, \mu X_3.\, \psi_3(X_1, X_2, X_3))) \,. \tag{5.3}$$

We rewrite ψ as

$$\begin{aligned} \psi &\equiv \mu X_1.\, \psi_1(X_1, \tau_1(X_1)) & (5.4)\\ \tau_1(X_1) &\doteq \nu X_2.\, \psi_2(X_1, X_2, \tau_2(X_1, X_2)) & (5.5)\\ \tau_2(X_1, X_2) &\doteq \mu X_3.\, \psi_3(X_1, X_2, X_3)\,, & (5.6) \end{aligned}$$

where each τ_i is monotonic in its arguments. As in the previous case, the computation starts with $X_1^0 = \emptyset, X_2^{00} = \mathcal{S}, X_3^{000} = \emptyset$[1]. Then, we iterate to compute the innermost fixpoint, resulting in $X_3^{00\omega}$. Now, the next approximation for X_2 is evaluated, $X_2^{01} = \psi_2(X_1^0, X_2^{00}, X_3^{00\omega})$, and we go back to the inner fixpoint X_3. We proceed computing the values $X_3^{01\omega}, X_2^{02}, \ldots, X_3^{0\omega\omega}, X_2^{0\omega}$, until we eventually reach the fixpoint value for X_2, $X_2^{0\omega}$—which, by definition, is equal to $\psi_2(X_1^0, X_2^{0\omega}, X_3^{0\omega\omega})$. Now, the outermost fixpoint X_1 is updated, $X_1^1 = \psi_1(X_1^0, X_2^{0\omega}, X_3^{0\omega\omega})$. We have that $X_1^0 \subseteq X_1^1$ and we are going to compute $X_2^{1\omega}$. Note that $X_2^{0\omega}$ and $X_2^{1\omega}$ are given by

$$\begin{aligned} X_2^{0\omega} &= \tau_1(X_1^0) & (5.7)\\ X_2^{1\omega} &= \tau_1(X_1^1) & (5.8) \end{aligned}$$

and, since τ_1 is monotonic, $X_2^{0\omega} \subseteq X_2^{1\omega}$. Unfortunately, this is relation is of no use, as X_2 is computed by a greatest fixpoint; so we have to initialize X_2 again, i.e., $X_2^{10} = \mathcal{S}$. Next we compute $X_3^{10\omega}$, but

$$\begin{aligned} X_3^{00\omega} &= \tau_2(X_1^0, X_2^{00}) & (5.9)\\ X_3^{10\omega} &= \tau_2(X_1^1, X_2^{01}) & (5.10) \end{aligned}$$

and by the monotonicity of τ_2 (since $X_1^0 \subseteq X_1^1$ and $X_2^{00} \subseteq X_2^{01}$) we see that $X_3^{00\omega} \subseteq X_3^{10\omega}$. As before, applying theorem 5.1, we see that we can avoid resetting X_3 to $\emptyset$ and use its previous value as a starting point, i.e., $X_3^{100} = X_3^{00\omega}$. Moreover, we have

$$\begin{aligned} X_2^{01} &= \psi_2(X_1^0, X_2^{00}, X_3^{00\omega}) & (5.11)\\ X_2^{11} &= \psi_2(X_1^1, X_2^{10}, X_3^{10\omega}) & (5.12) \end{aligned}$$

and from the previous relations, it follows that $X_2^{01} \subseteq X_2^{11}$. This implies that we can use the same trick again when computing $X_3^{11\omega}$, i.e., $X_3^{110} = X_3^{01\omega}$. And so on.

[1]We closely follow the example in [BCJ$^+$94].

We can repeatedly apply these monotonicity considerations. In general, when computing $X_3^{(k+1)m\omega}$, we can start from $X_3^{(k+1)m0} = X_3^{km\omega}$, which is generally larger than $\emptyset$, obtaining faster convergence. Following these ideas, the authors in [BCJ$^+$94] obtained the matrix of relationships for X_3 shown in figure 5.1, where the underlining and the arrows are ours. A new, delicate bound on time complexity were derived,

$$O(n^{d/2}) \text{ steps,}$$

which is the best time bound actually known among global algorithms for the full μ-calculus.

We note that this improved algorithm requires storing exponentially many intermediate results, roughly $O(n^{d/2})$. To see this, consider the previous example: when evaluating X_1^i, we must save the "frontier" of computed fixpoints $X_3^{ij\omega}$, $j = 1, 2, \ldots, \omega$, which will be successively used in the computation of X_1^{i+1}. This happens for each approximant X_2^{ij}, where j takes the values $1, 2$ $\ldots$, up to ω, which can be as big as n. This considerably reduces the applicability of such algorithm. In practical cases, the size of the BDD representation of the approximants is of the same order of magnitude of available memory. Hence, we cannot store too much previous values.

The same opinion can be found in [Eme97]. Here, the author observes that the algorithm by Browne et al. yields computational advantages only for formulae of alternation-depth at least 3, at the expense of an huge amount of memory. Since μ-calculus formulae of practical use have alternation-depth ≤ 2, the author concludes that, "it is not clear that there is ever an actual situation where the algorithm could be helpful". We build on this, trying to reduce memory consumption, but retaining some computational advantage for formulae with high alternation depth.

5.4 An improved algorithm

We are now ready to present the basic intuition behind our algorithm. As pointed out before, the algorithm by Browne et al. has time and space complexity of about $n^{d/2}$; we previously emphasized that the problem is especially with the space bound. We try to trade space for time. Our algorithm has space complexity d^2, while its time complexity is n^d, in the worst case. However, since we store more intermediate values than Emerson and Lei, we expect to perform better on the average case.

We now explain our basic idea. Return to the previous example. In figure 5.1 we have underlined the approximants of the form $X_3^{i\omega\omega}$. As j varies, the $X_3^{ij\omega}$'s constitutes the "frontier" values with respect to the i-th iteration of X_1. The previous algorithm required storing all these frontier values, updating them as i increases. Instead, we do only store one representative for each frontier: the underlined value. Consequently, the initial value $X_3^{(i+1)j0}$ of the following iteration is set to $X_3^{i\omega\omega}$ (instead of $X_3^{ij\omega}$). This is correct because, as can be

$$
\begin{array}{ccccccc}
\rightarrow & X_3^{\omega 0\omega} & \supseteq & X_3^{\omega 1\omega} & \supseteq & \cdots & \underline{X_3^{\omega\omega\omega}} \\
 & \rotatebox[origin=c]{90}{$\subseteq$} & & \rotatebox[origin=c]{90}{$\subseteq$} & & & \rotatebox[origin=c]{90}{$\subseteq$} \\
 & \vdots & & \vdots & & & \vdots \\
 & \rotatebox[origin=c]{90}{$\subseteq$} & & \rotatebox[origin=c]{90}{$\subseteq$} & & & \rotatebox[origin=c]{90}{$\subseteq$} \\
 & X_3^{\omega 01} & \supseteq & X_3^{\omega 11} & \supseteq & \cdots & X_3^{\omega\omega 1} \\
 & \rotatebox[origin=c]{90}{$\subseteq$} & & \rotatebox[origin=c]{90}{$\subseteq$} & & & \rotatebox[origin=c]{90}{$\subseteq$} \\
 & X_3^{\omega 00} & \supseteq & X_3^{\omega 10} & \supseteq & \cdots & X_3^{\omega\omega 0} \\
 & \| & & \| & & & \| \\
\vdots & \vdots & & \vdots & & & \vdots \\
 & \| & & \| & & & \| \\
\rightarrow & X_3^{10\omega} & \supseteq & X_3^{11\omega} & \supseteq & \cdots & \underline{X_3^{1\omega\omega}} \\
 & \rotatebox[origin=c]{90}{$\subseteq$} & & \rotatebox[origin=c]{90}{$\subseteq$} & & & \rotatebox[origin=c]{90}{$\subseteq$} \\
 & \vdots & & \vdots & & & \vdots \\
 & \rotatebox[origin=c]{90}{$\subseteq$} & & \rotatebox[origin=c]{90}{$\subseteq$} & & & \rotatebox[origin=c]{90}{$\subseteq$} \\
 & X_3^{101} & \supseteq & X_3^{111} & \supseteq & \cdots & X_3^{1\omega 1} \\
 & \rotatebox[origin=c]{90}{$\subseteq$} & & \rotatebox[origin=c]{90}{$\subseteq$} & & & \rotatebox[origin=c]{90}{$\subseteq$} \\
 & X_3^{100} & \supseteq & X_3^{110} & \supseteq & \cdots & X_3^{1\omega 0} \\
 & \| & & \| & & & \| \\
\rightarrow & X_3^{00\omega} & \supseteq & X_3^{01\omega} & \supseteq & \cdots & \underline{X_3^{0\omega\omega}} \\
 & \rotatebox[origin=c]{90}{$\subseteq$} & & \rotatebox[origin=c]{90}{$\subseteq$} & & & \rotatebox[origin=c]{90}{$\subseteq$} \\
 & \vdots & & \vdots & & & \vdots \\
 & \rotatebox[origin=c]{90}{$\subseteq$} & & \rotatebox[origin=c]{90}{$\subseteq$} & & & \rotatebox[origin=c]{90}{$\subseteq$} \\
 & X_3^{001} & \supseteq & X_3^{011} & \supseteq & \cdots & X_3^{0\omega 1} \\
 & \rotatebox[origin=c]{90}{$\subseteq$} & & \rotatebox[origin=c]{90}{$\subseteq$} & & & \rotatebox[origin=c]{90}{$\subseteq$} \\
 & X_3^{000} & \supseteq & X_3^{010} & \supseteq & \cdots & X_3^{0\omega 0}
\end{array}
$$

Figure 5.1: Monotonicity relations for X_3

seen from the figure (by transitivity), for all j

$$X_3^{i\omega\omega} \subseteq X_3^{(i+1)j\omega} \ . \tag{5.13}$$

Before delving into the details of the algorithm, we first need some formal definitions.

5.4.1 Formulae, subformulae and so on

A formula g is a *subformula* of a formula f if (the parse tree of) g appears somewhere in (the parse tree of) f. g is a *proper subformula* of f provided it is a subformula of f and $g \neq f$. It is easy to see that the "subformula" binary relation is indeed a partial order. *Top-level* (or *immediate*) subformulae are maximal proper subformulae. Finally, the "enclosing" relation is defined to be the inverse of the "subformula" relation: f is an *enclosing formula* of g if g is a subformula of f.

For example, in the μ-calculus formula

$$f \doteq \mu\, x.\, p \vee \mathbf{EX}x$$

there are 5 subformulae, which are f itself, $p \vee \mathbf{EX}x$, p, $\mathbf{EX}x$ and x. All but f are also proper subformulae, and $p \vee \mathbf{EX}x$ is a top-level subformula (in this case, the only one).

Sometimes we need to restrict our attention only to fixpoint formulae. In the sequel, we will use $\sigma \in \{\mu, \nu\}$ as a generic fixpoint operator; σ satisfies:

$$\bar{\sigma} = \begin{cases} \nu & \text{if } \sigma = \mu \\ \mu & \text{otherwise} \end{cases} \qquad \text{and} \qquad \bar{\bar{\sigma}} = \sigma \ . \tag{5.14}$$

Given a formula of the form $\sigma\, x.\, f$, *immediate σ-subformulae* are top-level subformulae of the form $\sigma\, y.\, g$. Similarly, *immediate $\bar{\sigma}$-subformulae* are top-level subformulae of the form $\bar{\sigma}\, y.\, g$. We denote the transitive closure of the "immediate σ-subformula" relation as $\prec_\sigma$. Alternatively, the *δ-subformula* relation $\prec_\delta$, for $\delta = \sigma, \bar{\sigma}$, can be defined as

$$\delta\, y.\, g \prec_\delta \sigma\, x.\, f \iff \delta\, y.\, g \text{ is a proper subformula of } f. \tag{5.15}$$

As usual, we denote with $\preceq_\sigma$ the reflexive closure of $\prec_\sigma$. Moreover, we define $\prec \doteq \prec_\mu \cup \prec_\nu$. *Enclosing σ-formulae* are defined in the obvious way. In the following, we will omit the body of fixpoint operators, e.g., we will write $\sigma\, y \prec_\sigma \sigma\, x$ instead of $\sigma\, y.\, g \prec_\sigma \sigma\, x.\, f$.

For example, consider the formula f_1 defined as

$$\begin{aligned} f_1 &\doteq \mu\, X_1.\, (\varphi_1(X_1) \vee f_2) \\ f_2 &\doteq \nu\, X_2.\, (\varphi_2(X_2) \wedge f_3) \\ f_3 &\doteq \mu\, X_3.\, \varphi_3(X_3) \ . \end{aligned}$$

Then, the following relations hold:

$$\begin{aligned} f_1 &\preceq_\mu f_1 \\ f_2 &\prec_\nu f_1 \\ f_3 &\prec_\mu f_1, f_2 \ . \end{aligned}$$

A sequence of fixpoint formulae of the same type $\sigma\, x_1 \prec_\sigma \sigma\, x_2 \prec_\sigma \cdots \prec_\sigma \sigma\, x_n$ can be grouped into an equivalence class when no fixpoint formula of the form $\bar\sigma\, y$ appears between x_n and x_1, i.e., there is no formula $\bar\sigma\, y$ s.t. $\sigma\, x_1 \prec_\sigma \bar\sigma\, y \prec_{\bar\sigma} \sigma\, x_n$. If this is the case, for every x_i we define its equivalence class as

$$[x_i] \doteq \{x_1, \ldots, x_n\} \ . \tag{5.16}$$

It is easy to see that the equivalence relation corresponding to those equivalence classes groups together fixpoint subformulae of the same type when no alternation of fixpoint occurs inside them, i.e., they have the same *alternation depth* in the sense of Emerson and Lei.

For example, in the formula f_1 defined as

$$\begin{aligned} f_1 &\doteq \mu\, X_1.\, \varphi_1(X_1, f_2(X_1)) \\ f_2(X_1) &\doteq \mu\, X_2.\, \varphi_2(X_1, X_2, f_3(X_1)) \\ f_3(X_1) &\doteq \mu\, X_3.\, \varphi_3(X_1, X_3) \ , \end{aligned}$$

all the X_i's belong to the same equivalence class, i.e.,

$$[X_1] = [X_2] = [X_3] = \{X_1, X_2, X_3\} \ .$$

We say that $\delta\, y.\ g$ *depends on* an enclosing formula $\delta'\, x.\ f \succ \delta\, y.\ g$ if x is free in g. We overload this relation and with "depends on" we denote also its transitive closure. In the previous example, both f_2 and f_3 depend on f_1, while f_3 does not depend on f_2.

5.4.2 The σ-level of fixpoint formulae

We now introduce the notion of σ*-level* of a σ-subformula. Intuitively, given a formula f, the σ-level of a σ-subformula g of f counts approximately the number of "genuine double alternations" of fixpoints in a depth-first exploration in the parse tree from f to g. "Double" means that we count alternations in pairs, e.g., if a formula is of alternation depth d, the σ-level of its innermost σ-formula g is roughly $d/2$. "Genuine" means that inner fixpoint formulae must depend on outer ones. We need this definition as for each relational variable Q, the σ-level of its binding formula will determine the amount of past memory that Q has on its previous values.

More formally, consider a formula f. The σ*-level* of its fixpoint subformulae is determined according to these two rules:

- topmost μ and ν-subformulae g have σ-level equal to one: σ-level$(g) = 1$.

- If $g = \sigma\, x$ is a subformula of σ-level$(g) = n$, then for all immediate σ-subformulae $g' = \sigma\, x'$:
  - σ-level$(g') = n + 1$ if x' depends on some $\bar{\sigma}\, y \prec_{\bar{\sigma}} \sigma\, x$ and y depends on some $x_i \in [x]$.
  - σ-level$(g') = n$, otherwise. In particular σ-level$(g') = n$ if $x' \in [x]$.

Note that, if a formula has alternation depth d, the maximum σ-level of its subformulae is $(d + 1)/2$.

The *relative σ-level* of a δ-subformula f wrt an enclosing δ'-subformula g is the σ-level of f when g is considered a topmost formula, i.e., without any enclosing fixpoint formula. The relative σ-level of f wrt f itself is 1 by definition. We will also speak about the σ-level of a relational variable, intending the σ-level of its binding fixpoint formula.

For example, in the formula

$$\nu\, X_1.\, \psi_1(X_1, \mu\, X_2.\, \psi_2(X_1, X_2, \nu\, X_3.\, \psi(X_1, X_2, X_3)))$$

X_1 has ν-level 1, X_2 has μ-level 1 and X_3 has ν-level 2, while in the formula

$$\mu\, X_1.\, \psi_1(X_1, \mu\, X_2.\, \psi_2(X_1, X_2, \mu\, X_3.\, \psi(X_1, X_2, X_3)))$$

the σ-level is 1 for all X_1, X_2, X_3. Finally, also in the formulae

$$\nu\, X_1.\, \psi_1(X_1, \mu\, X_2.\, \psi_2(X_1, X_2, \nu\, X_3.\, \psi(X_1, X_3))),\ \text{and}$$
$$\nu\, X_1.\, \psi_1(X_1, \mu\, X_2.\, \psi_2(X_2, \nu\, X_3.\, \psi(X_1, X_2)))$$

all σ-level are equal to 1, since there is no real double alternation, as X_3 does not depend on X_2 in the first formula and X_2 does not depend on X_1 in the second.

5.4.3 The core of the algorithm

The core of our algorithm is shown in figure 5.3. Boolean operations and next-state operators are handled as usual, and will not be shown. We will informally explain the behaviour of our algorithm through an example. Consider the formula

$$\begin{aligned} \psi &\equiv \mu\, X_1.\, \psi_1(X_1, \tau_1(X_1)) && (5.17)\\ \tau_1(X_1) &\doteq \nu\, X_2.\, \psi_2(X_1, X_2, \tau_2(X_1, X_2)) && (5.18)\\ \tau_2(X_1, X_2) &\doteq \mu\, X_3.\, \psi_3(X_1, X_2, X_3)\ . && (5.19)\end{aligned}$$

We associate to each relational variable X_i an array val(X_i) of n past values, where $n = \sigma$-level(X_i), previously introduced[2]. In general, val$(X_i)(1)$ is the most recent value and val$(X_i)(n)$ the oldest one. In our example,

$$\sigma\text{-level}(X_1) = 1, \quad \sigma\text{-level}(X_2) = 1, \quad \text{and} \quad \sigma\text{-level}(X_3) = 2\ , \qquad (5.20)$$

[2]We identify X_i with the whole binding fixpoint formula $\sigma X_i(\ldots)$ when considering its σ-level.

so X_1 and X_2 remember only one past value, while X_3 remembers two. The first step of the algorithm involves a global initialization (once and for all) of the "topmost" values $\mathrm{val}(X_i)(n)$:

$$\mathrm{val}(X_1)(1) \doteq \emptyset \tag{5.21}$$
$$\mathrm{val}(X_2)(1) \doteq \mathcal{S} \tag{5.22}$$
$$\mathrm{val}(X_3)(2) \doteq \emptyset\ . \tag{5.23}$$

Then, the computation begins from the topmost fixpoint, i.e., X_1. The initial value of X_1 is read from $\mathrm{val}(X_1)$:

$$X_1^0 \doteq \mathrm{val}(X_1)(1) = \emptyset\ . \tag{5.24}$$

The PreInit operation (shown in figure 5.2) is called before each iteration and acts on inner fixpoints of *opposite type* and of σ-level ≥ 2; its purpose is to initialize "inner" locations with past values. As the only inner opposite fixpoint of X_1 is X_2, but $\sigma\text{-level}(X_2) = 1$, no initialization is triggered by X_1. Then, the control passes to X_2:

$$X_2^{00} \doteq \mathrm{val}(X_2)(1) = \mathcal{S}\ . \tag{5.25}$$

As $\sigma\text{-level}(X_3) = 2$, X_2 pre-inits X_3:

$$\mathrm{val}(X_3)(1) \doteq \mathrm{val}(X_3)(2) = \emptyset\ . \tag{5.26}$$

Now, the computation of $X_3^{00\omega}$ takes place. The initial value is fetched from $\mathrm{val}(X_3)$:

$$X_3^{000} \doteq \mathrm{val}(X_3)(1) = \emptyset\ , \tag{5.27}$$

and we iterate until convergence. At convergence (actually, at each iteration), the fixpoint value is stored again in $\mathrm{val}(X_3)(1)$:

$$\mathrm{val}(X_3)(1) \doteq X_3^{00\omega}\ , \tag{5.28}$$

and the control passes to X_2, which computes X_2^{01}. A new iteration for X_2 starts and the init operation triggers the reset of X_3:

$$\mathrm{val}(X_3)(1) \doteq \mathrm{val}(X_3)(2) = \emptyset\ , \tag{5.29}$$

so that the following computation of $X_3^{01\omega}$ must start from scratch (and we cannot do better):

$$X_3^{010} \doteq \mathrm{val}(X_3)(1) = \emptyset\ . \tag{5.30}$$

This repeats until convergence of X_2. At that time, the values of $\mathrm{val}(X_3)(1)$ and $\mathrm{val}(X_3)(1)$ are:

$$\mathrm{val}(X_3)(1) \doteq X_3^{0\omega\omega} \tag{5.31}$$
$$\mathrm{val}(X_2)(1) \doteq X_2^{0\omega}\ . \tag{5.32}$$

Now the control goes to X_3, having computed X_3^1. At the end of each iteration, the PostUpdate operation is called to save previously computed values. It acts on inner fixpoints of the *same type* and σ-level ≥ 2. (Note that X_2 and X_3 trigger no update, as they do not have inner fixpoints of type equal to their own.) For X_1, the only updated value is X_3. The behaviour of PostUpdate is somewhat complementary to PreInit, as the information goes in the opposite direction:

$$\mathrm{val}(X_3)(2) \doteq \mathrm{val}(X_3)(1) = X_3^{0\omega\omega} \ . \tag{5.33}$$

A new iteration of X_1 starts, hence a new iteration of X_2, which inits X_3:

$$\mathrm{val}(X_3)(1) \doteq \mathrm{val}(X_3)(2) = X_3^{0\omega\omega} \ , \tag{5.34}$$

so the computation of $X_3^{10\omega}$ starts from $X_3^{0\omega\omega}$ (and not from $\emptyset$), as expected:

$$X_3^{100} \doteq \mathrm{val}(X_3)(1) = X_3^{0\omega\omega} \ . \tag{5.35}$$

Moreover, $X_3^{0\omega\omega}$ is the starting value also for the computation of the following fixpoints $X_3^{11\omega}, X_3^{12\omega}, \ldots$, until $X_3^{1\omega\omega}$ is computed and stored for later use:

$$\mathrm{val}(X_3)(1) \doteq X_3^{1\omega\omega} \ . \tag{5.36}$$

Then the computation proceeds on the same line, until X_1 finally converges. At that time, the memory locations would hold the following values:

$$\begin{aligned} \mathrm{val}(X_1)(1) &= X_1^{\omega} && (5.37)\\ \mathrm{val}(X_2)(1) &= X_2^{\omega\omega} && (5.38)\\ \mathrm{val}(X_3)(2) &= X_3^{\omega\omega\omega} \ , && (5.39) \end{aligned}$$

and the computation terminates.

Note that, when a fixpoint is nested in a fixpoint of the same type, the PreInit and PostUpdate operations avoid resetting the inner fixpoint at each outer iteration. Hence, our algorithm make use of the aforementioned observation by Emerson & Lei.

5.4.4 Space considerations

In the worst case, a formula of alternation-depth d has only strict alternations, e.g., is in the form (assuming d odd)

$$\sigma X_1 \bar{\sigma} X_2 \sigma X_3 \ldots \bar{\sigma} X_{d+1}$$

and for each X_i we store $\lceil i/2 \rceil$ values. Then, our worst case space bound is:

$$\sum_{i=1}^{d+1} \lceil i/2 \rceil \leq 2 \sum_{j=1}^{\lceil (d+1)/2 \rceil} j = O(d^2) \tag{5.40}$$

```
Algorithm 5.1: The PreInit operation.
input : A $\sigma$-formula $\sigma\, X$.
result: Value $\mathrm{val}(Y)(i)$ is initialized for some subformulae $\bar\sigma\, Y$.
For all $\bar\sigma\, Y \prec_{\bar\sigma} \sigma\, X$ of relative $\bar\sigma$-level $i$ wrt $X$ and $\bar\sigma$-level$(\bar\sigma\, Y) \geq 2$,
$\mathrm{val}(Y)[\mathrm{i}] := \mathrm{val}(Y)[\mathrm{i+1}]$ .
```

```
Algorithm 5.2: The PostUpdate operation.
input : A $\sigma$-formula $\sigma\, X$.
result: Value $\mathrm{val}(X')(i)$ is initialized for some subformulae $\sigma\, X'$.
For all $\sigma\, X' \prec_{\sigma} \sigma\, X$ of relative $\sigma$-level $i$ wrt $X$ and $\sigma$-level$(\sigma\, X') \geq 2$,
$\mathrm{val}(X')[\mathrm{i+1}] := \mathrm{val}(X')[\mathrm{i}]$ .
```

Figure 5.2: The PreInit and PostUpdate operations.

1. *Global initialization.*

   For all formulae $\sigma\, Z$ of σ-level$(Z) = n$,

$$\mathrm{val}(Z)[n] = \begin{cases} \emptyset & \text{if } \sigma = \mu \\ \mathcal{S} & \text{otherwise.} \end{cases}$$

2. *Fixpoint computation.*

```
Algorithm 5.3: The fixpoint algorithm.
input : A $\sigma$-formula $\sigma\, X$. $f$, denoting the
        monotone predicate transformer $\tau(W)$.
result: The fixpoint value of $f(X)$.
$W := \mathrm{val}(X)[1]$;
repeat
    $W := W'$;
    PreInit($\sigma\, X$);
    $W' := \tau(W')$;
    PostUpdate($\sigma\, X$);
    $\mathrm{val}(X)[1] := W'$;
until $W \neq W'$;
```

Figure 5.3: The fixpoint algorithm.

(with equality when d is odd).

Finally, we can compare the space complexity of our algorithm with respect to Emerson and Lei's and Browne's et al., as shown in figure 5.4. It is worth comparing the polynomial space complexity of ours and Emerson and Lei's algorithms to the exponential space complexity of Browne's et al. algorithm.

Algorithm	Space complexity
Emerson and Lei	$O(d)$
Our algorithm	$O(d^2)$
Browne et al.	$O(n^d)$

Figure 5.4: Space complexity for μ-calculus model-checking algorithms.

Chapter 6

Implementation

In this chapter we discuss implementation issues. We describe how we extended the NuSMV model-checker to include both a μ-calculus model-checking engine and the specification logics μ-calculus and ω-**CTL**. We refer to actual source files, pointing out relevant changes and/or additional code. Moreover, we also explain how efficient model-checking for certain μ-calculus formulae can be performed using an *incremental representation* for the approximants of the required fixpoint.

The rest of the chapter is organized as follows. In Section 6.1 we point out which NuSMV modules have been modified and/or extended. In particular, the `parser` module has been extended to accommodate the syntax of μ-calculus and ω-**CTL** formulae. Also minor changes aiming at eliminating various memory leaks (that we found in NuSMV) are mentioned. In Section 6.2 we describe the new module `mucalculus`, where all the algorithms and utility functions have been implemented. Finally, to make things concrete, in Section 6.3 we present practical examples of μ-calculus and ω-**CTL** formulae in the concrete syntax of (our extension of) NuSMV.

6.1 Modified NuSMV modules

The NuSMV open-source symbolic model-checker [CCG$^+$02] is an huge piece of software. It consists of roughly 180 thousands lines of code, but its modular structure allows the addition of further functionalities with relatively little effort. Given the complexity of its structure, we will not give a detailed description of its modules here. We refer the interested reader to the user manual [CCJ$^+$b] and to the programmer manual [CCJ$^+$a].

As the code of NuSMV is freely available, we had the occasion of directly integrating our new functionalities into it. We modified some existing modules (like the `parser`) and we added the new `mucalculus` module. Moreover, we made some (minor) little fixes, mainly addressing secondary memory leaks. In this section we address modified modules, while the new `mucalculus` module is described in the following Section 6.2.

6.1.1 Main changes

Module `compile`

`compileCmd.c`. In function `CommandFlattenHierarchy` a call is made to `FlatHierarchy_get_muspec(mainFlatHierarchy)` to include μ-calculus specification into the model.

`compileFlatten.c`. Implemented correct handling of μ-calculus formulae.

`compileUtil.c`. Added handling of relational variables in `Compile_pop_distrib_ops_recurse`.

`SymbCache.c`. Modified `SymbCache_is_symbol_input_var` to allow the input variable `{_process_selector_}` to be used into specifications.

`CheckerCore.c`. Type checking for the new μ-calculus and ω-**CTL** operators that we introduced has been considered, but no type-checking is actually done.

Module `parser`

`grammar.y`. This is the Yacc grammar. We extended it adding rules for parsing our new μ-calculus and ω-**CTL** expressions. See figures 6.1 and 6.2 for our new productions.

`input.l`. This is the Lex input source, where we added new tokens, used when parsing μ-calculus and ω-**CTL** expressions. In particular, there is a flag, `in_regexpr`, that is set when parsing a ω-regular expression: when the flag is set, the lexer returns `TOK_REGBINPLUS` when reading '`+`' and `TOK_REGSEMI` when reading '`;`', instead of `TOK_PLUS` and `TOK_PLUS`, resp. That is, we overloaded input tokens depending on the context, which turns to be useful during parsing.

`symbols.h`. In this file there are all the definitions of the identifier used during parsing. We added identifiers for μ-calculus and ω-**CTL** operators.

`parserUtil.c`. Added calls to finalizing functions `free_relvar_stacks` and `free_mu_nu_stack` (called at the end of parsing).

Module `node`

`node.h`. The data structure `node_t` has been extended with an new field `void *priv_data`. However, for our purposes, the actual type of `priv_data` is `extra_ptr`. The purpose of the field `priv_data` is to store extra data about

```
⋮

%left <lineno> [...] TOK_MUSPEC TOK_MUOP TOK_NUOP
     TOK_RELVAR [...]

⋮

%left  <lineno> TOK_REGBINPLUS
%left  <lineno> TOK_REGSEMI
%left  <lineno> TOK_REGINFOFT TOK_REGSTAR TOK_REGPLUS
     TOK_REGOMEGA

⋮

%type <node> mu_relational_variable new_mu_relational_variable
             new_nu_relational_variable
%type <node> mu_fixop_expr
%type <node> mu_expression
%type <node> regular_expr regular_expr_list ext_ctl_expr
%type <node> [...] muspec [...]

⋮
%%
⋮

primary_expr :
                [...]
              | mu_relational_variable
                [...]
mu_fixop_expr :
                implies_expr
              | ext_ctl_expr
              | TOK_MUOP new_mu_relational_variable
    mu_fixop_expr { [...] }
              | TOK_NUOP new_nu_relational_variable
    mu_fixop_expr { [...] }
                ;
```

Figure 6.1: Yacc grammar for μ-calculus.

```
regular_expr_list:
    regular_expr { [...] }
  | regular_expr TOK_COMMA regular_expr_list { [...] }
  ;

regular_expr:
    TOK_LB
       { in_regexpr = 0; } mu_expression
    TOK_RB { in_regexpr = 1; } { [...] }
  | TOK_LP regular_expr TOK_RP                { [...] }
  | regular_expr TOK_REGBINPLUS regular_expr  { [...] }
  | regular_expr TOK_REGSEMI regular_expr     { [...] }
  | regular_expr TOK_REGPLUS                  { [...] }
  | regular_expr TOK_REGSTAR                  { [...] }
  | regular_expr TOK_REGOMEGA                 { [...] }
  | TOK_LCB regular_expr_list TOK_RCB TOK_REGINFOFT { [...] }
       ;

ext_ctl_expr:
     TOK_EF TOK_LB
{ in_regexpr = 1; } regular_expr TOK_COMMA { in_regexpr = 0; }
          mu_expression TOK_RB { [...] }
  |  TOK_AF TOK_LB
{ in_regexpr = 1; } regular_expr TOK_COMMA { in_regexpr = 0; }
          mu_expression TOK_RB { [...] }
  |  TOK_EG TOK_LB
{ in_regexpr = 1; } regular_expr TOK_COMMA { in_regexpr = 0; }
          mu_expression TOK_RB { [...] }
  |  TOK_AG TOK_LB
{ in_regexpr = 1; } regular_expr TOK_COMMA { in_regexpr = 0; }
          mu_expression TOK_RB { [...] }
  |  TOK_EU TOK_LB
{ in_regexpr = 1; } regular_expr TOK_COMMA { in_regexpr = 0; }
          mu_expression TOK_COMMA mu_expression TOK_RB
               { [...] }
  |  TOK_AU TOK_LB
{ in_regexpr = 1; } regular_expr TOK_COMMA { in_regexpr = 0; }
          mu_expression TOK_COMMA mu_expression TOK_RB
               { [...] }
       ;
```

Figure 6.2: Yacc grammar for ω-**CTL** and ω-regular expressions.

formulae (as explained below, see 6.2, on page 75), the main purpose being caching formula's value during model-checking.

`node.c`. The functions `new_lined_node`, `find_node` and `insert_node` are been modified to reflect the added field `priv_data`. Moreover, the function `new_node` has been re-factored and a call of `free_assoc` as been introduced into `node_quit` to eliminate a memory leak.

`printers/PrinterWffCore.c`. The function `printer_wff_core_print_node` has been modified to correctly print the new operators of μ-calculus, ω-**CTL** and ω-regular expressions.

Module `prop`

`propProp.c` The function `Prop_get_expr_core` has been extended to call the new function `mucalculus_convert_muformula_to_core` when dealing with μ-calculus properties.

Module `main`

`smMisc.c`. Code has been added in order to verify also μ-calculus specifications when verifying **CTL**/**LTL**/**PSL** specifications.

`smInit.c`. The μ-calculus module is properly setup and shutdown. Also fixed various memory leaks.

Module `mc`

We added model-checking capabilities for the μ-calculus. In particular, we created a new file `mcMu.c`, implementing our model-checking algorithm. We based our implementation on the existing `mcMc.c`, from which we borrowed the coding-style. The central function is `fix`, which implements the algorithm previously described in Section 5.4. Moreover, from `mcMc.c` we borrowed a significant practical improvement when computing fixpoints: the main idea is that in some cases it is not necessary to iterate using the whole approximant X_i, maybe $X_i \setminus X_{i+1}$ (which is smaller) may suffice. This allows for a faster algorithm (convergence is not reached with fewer steps, but each step takes shorter). We will return to this topic in detail in Section 6.1.2.

Finally, we slightly modified the file `mcMc.c`, including a counter of iterations to reach convergence (for comparing the algorithms) and a debugging mechanism, used to test our μ-calculus model-checker against the one in NuSMV on **CTL** formulae.

6.1.2 Efficient incremental model-checking

We now show how sometimes we can speed up fixpoint computation. Consider the computation of a least fixpoint, whose chain of approximants is

$$\emptyset = A_0 \subseteq A_1 \subseteq A_2 \subseteq \cdots \ , \tag{6.1}$$

where $A_i \doteq f^i(\emptyset)$. Now we define the *difference* D_n between A_n and A_{n-1}:

$$D_n \doteq A_n \cap \overline{A_{n-1}} \ . \tag{6.2}$$

When A_n and A_{n-1} differ by few states, D_n is rather small. So, if we would be able to compute $A_{n+1} \doteq f(A_n)$ from A_n and a simple function of D_n, we would speed up computation. In some cases this is possible. Consider the following

Theorem 6.1. *Let f be a monotonic function from $\mathcal{S}$ to $\mathcal{S}$ and let $A_n \doteq f^n(\emptyset)$. As before, let D_n be*

$$D_n \doteq A_n \cap \overline{A_{n-1}} \ .$$

Then, for all n, we have that:

1. *$A_n = D_n \cup A_{n-1}$ and $D_n \cap A_{n-1} = \emptyset$,*
2. *$A_n = \bigcup_{1 \le m \le n} D_m$, and*
3. *$f(A_n) \supseteq A_n \cup f(D_n)$.*

Proof. We split the proof by cases.

1. Directly follows from the definition.
2. We have to show that $A_n = \bigcup_{1 \le m \le n} D_m$, for all n. We prove this by induction on n:

   (a) Basis, $n = 1$. As $A_0 = \emptyset$, $D_1 = A_1$ and this point is complete.

   (b) Assume $A_n = \bigcup_{1 \le m \le n} D_m$. Then,

$$\begin{aligned} A_{n+1} &= D_{n+1} \cup A_n = \text{ (by point 1.)} \\ &= D_{n+1} \cup \bigcup_{1 \le m \le n} D_m = \text{ (induction hyp.)} \\ &= \bigcup_{1 \le m \le n+1} D_m \ , \end{aligned}$$

   hence $A_{n+1} = \bigcup_{1 \le m \le n+1} D_m$ and this part of the proof is complete.

3. To show that $f(A_n) \supseteq A_n \cup f(D_n)$, we note that the following relations hold:

$$f(A_n) = f(\bigcup_{1 \le m \le n} D_m) = \text{ (by point 2.)}$$

$$
\begin{aligned}
&= f\Big(\Big(\bigcup_{1\le m\le n-1} D_m\Big) \cup D_n\Big) = (*) \\
&\supseteq f\Big(\bigcup_{1\le m\le n-1} D_m\Big) \cup f(D_n) = \text{ (by monotonicity.)} \\
&= f(A_{n-1}) \cup f(D_n) = \\
&= A_n \cup f(D_n) \ ,
\end{aligned}
$$

hence $f(A_n) \supseteq A_n \cup f(D_n)$, and the theorem is proved.

□

From the previous theorem, it follows that we can always under-approximate $f(A_n)$ with $A_n \cup f(D_n)$. We would like to know when this approximation is indeed exact, i.e., when we have equality:

$$f(A_n) = A_n \cup f(D_n) \ . \tag{6.3}$$

Note that in the crucial step (*) we only used monotonicity, and continuity does not help since the D_n's do not form a chain. However, if we put some restrictions on f, we can achieve equality in (*). Ideally, we would require that, for *arbitrary* subsets B_1, B_2 of $\mathcal{S}$,

$$f(B_1 \cup B_2) = f(B_1) \cup f(B_2) \ . \tag{6.4}$$

It is easy to see that this suffices for equality in (*). A sufficient condition for f to satisfy equation 6.4 is that $f(X)$ is the predicate transformer of a formula in *positive normal form* (i.e., negation only at the atomic level), X does not appear inside any universal next-state operator $\mathbf{AX}(\cdot)$ and whether $\wedge$ is used, at least one of the two operands must be a closed formula[1]. Under these conditions, we can use the incremental representation D_n and we can compute $f(A_n)$ by $A_n \cup f(D_n)$.

For maximal fixpoints, an analogous condition can be given. First, we use the definition:

$$D'_n \doteq A_n \cup \overline{A_{n-1}} \ , \tag{6.5}$$

which satisfies the following dual theorem (which we do not prove):

Theorem 6.2. *Let f be a monotonic function from $\mathcal{S}$ to $\mathcal{S}$ and let $A_n \doteq f^n(\mathcal{S})$. Then, for all n, we have that:*

1. *$A_n = D'_n \cap A_{n-1}$ and $D'_n \cup A_{n-1} = \mathcal{S}$,*
2. *$A_n = \bigcap_{1\le m\le n} D'_m$, and*
3. *$f(A_n) \subseteq A_n \cap f(D'_n)$.*

[1]This is essentially what we proved in Chapter 2, theorem 2.2.

Hence, to obtain the equality

$$f(A_n) = A_n \cap f(D'_n) \tag{6.6}$$

we require that $f(X)$ is the predicate transformer of a formula in positive normal form, X does not appear inside any existential next-state operator $\mathbf{EX}(\cdot)$ and whether $\vee$ is used, at least one of the two operands must be a closed formula.

Concluding, for some restricted class of formulae we can speed up fixpoint computation by using the incremental representation D_n (D'_n, resp.), which, from a practical point of view, results in a decrease of execution time of a factor 3 or 4.

6.1.3 Minor changes

Module `compile`

- `compileCheck.c`. We added the functions `free_check_constant_hash`, `free_global_assign_hash`, `free_assign_hash`, adding appropriate calls in the function `Compile_CheckAssigns`.
- `compileCmd.c`. Function calls to `free_check_constant_hash` and `cmp_struct_free` added in function `Compile_quit`.
- `compileStruct.c`. Added function `cmp_struct_free`.
- `PredicateNormaliser.c`. Fixed memory leaks.

Module `fsm`

Added call to `ClusterList_destroy` into `FsmBuilder.c` to fix a memory leak. Also fixed a memleak in `SexpFsm.c`.

Module `option`

Added `free_options` in `opt.h` and `opt.c`, solving a memory leak.

Module `utils`

Added `string_free` in `ustring.c` solving a memory leak associated with unfreed strings in the string manager.

Module `trans`

Solved some memory leak into `bdd/ClusterList.c`, but others remain to be fixed.

6.2 The new module `mucalculus`

Most of new code has been packaged in the new `mucalculus` module.

`mucalculus.h`. In this file there are all the function declarations used throughout other modules. Moreover, useful data structures are declared, which are `relvar_` for relational variables and `extra_` for extra data attached to formulae. In particular, the most important fields of `relvar_` are:

- `enum type`: whether this is a maximal or minimal fixpoint operator.
- `int uniq_index`: unique global index for relational variables, useful to distinguish relational variables with the same name.
- `bdd_ptr* val`: pointer to the first item of an array of past values (represented as ROBDDs), it is the implementation of the array val$(\cdot)$ presented earlier in 5.4.3.
- `int sigma_level`: σ-level of relational variable, as defined in 5.4.2.
- `node_ptr subf`: pointer to the first fixpoint subformula of the same type.
- `node_ptr subf2`: pointer to the first fixpoint subformula of the opposite type.
- `node_ptr nextf`: "next" pointer in the list of fixpoint formulae of the same type and with the same syntactic nesting depth.

The most important fields of `extra_` are:

- `lsList rel_vars`: list of free relational variables appearing into this formula (i.e., the formula this extra value refers to).
- `bdd_ptr* prev_rv_vals`: for each of the previous free relational variables, there is an element in this array storing its previous value (no waste of memory as ROBDDs are stored by reference); it is used to determine if a formula needs to be re-evaluated, which happens when there is some free relational variables whose actual value does not match the one in `prev_rv_vals`.
- `bdd_ptr last_val`: cached ROBDD representing the value of this formula; when a formula is being to be re-evaluated, but no relational variable it depends on is actually changed, this value is returned.

`mucalculus.c`. Utility functions common to other modules are packaged together into this source file. In particular, it contains the stack manipulating routines `stack_push`, `stack_pop`, `stack_top`; the generic list finding routine `appears_in_list`; the routine `mucalculus_convert_muformula_to_core`, that converts a hybrid μ-calculus formula, possibly containing also **CTL** and ω-**CTL** operators, into a pure μ-calculus formula, as we now explain.

In parsing, we allow free mixing of **CTL**, ω-**CTL** and μ-calculus operators. But, for the purpose of model-checking, we need a pure μ-calculus formula. This is obtained applying the translations from **CTL** and ω-**CTL** to

μ-calculus presented before (see sections 2.5 and 4.4, respectively). Moreover, we need to enrich formulae with information concerning the nesting and the σ-level of fixpoint operators and relational variables. All this stuff is done in the `mucalculus_convert_muformula_to_core` (called from `propProp.c` in module `prop`), which calls the following functions, in the appearing order:

- `mucalculus_conv_ext_ctl_to_mu` (described in `regexpr.c`),
- `mucalculus_conv_ctl_to_mu` (described in `ctl_to_mu.c`),
- `mucalculus_enrich_spec` (described in `extra.c`).

`regexpr.c`. This is where all processing of ω-regular expressions and ω-**CTL** is done. Main functions are:

- `mucalculus_conv_ext_ctl_to_mu`: convert ω-**CTL** formulae into the μ-calculus, calling the functions that follows,
- `mucalculus_conv_ext_ctl_eg_reg`,
- `mucalculus_conv_ext_ctl_eg`,
- `mucalculus_conv_ext_ctl_eu_reg`,
- `mucalculus_conv_ext_ctl_eu`: all these functions behave as described previously in 4.4. They depend on the utility functions that follow.
- `regexpr_equal`: determine whether two ω-regular expressions are syntactically the same.
- `regexpr_myfind_node(_int)`: create a new syntactical node, implementing usual simplifications for regular expressions, e.g., $\varepsilon \cdot x = x \cdot \varepsilon = x$ and others.
- `regexpr_is_epsilon_free`: determine whether a ω-regular expression is ε-free, implementing the predicate shown before in 4.1.
- `regexpr_make_epsilon_free`: fins the most similar ε-free expression to a given expression, as shown in 4.1.
- `regexpr_fin_int`,
- `regexpr_inf_int`: these two functions return the finite and infinite part, respectively, of a ω-regular expression, as shown in 4.1 and 4.1, respectively.
- `regexpr_put_in_normalform`: put a ω-regular expression in normal form, according to what shown in 4.1.

`relvar.c`. This file contains creators and destructors for relational variables, along with various getters and setters for the `relvar_` data structure, defined in `mucalculus.h`. Of its own importance are the two functions below:

- `relvar_init_val`,
- `relvar_update_val`: these functions implement the PreInit and PostUpdate operations described early in 5.4.3.

`ctl_to_mu.c`. The principal function is `mucalculus_conv_ctl_to_mu`, which translates **CTL** operators in μ-calculus, using the method previously defined in Section 2.5. As a side effect, various fields of relational variables are set, like `subf`, `nextf` and others.

`extra.c`. This source files defines a lot of functions for the `extra_` data structure, most being getters, setters and alike. Most notable are the following functions:

- `mucalculus_add_extra_rec`: this function allocates and initializes all the extra data associated with formulae. In particular, for each subformula, free relational variables are gathered and recorded into the `rel_vars` field.
- `mucalculus_augment_spec_rec`: this function completes the initialization of relational variables, filling in the fields `subf2` and `sigma_level`.
- `mucalculus_enrich_spec`: this function calls the two above functions (in the order they appear), completing the enrichment process of μ-calculus formulae.

`parse.c`. In this file are defined utility functions manipulating the stack of relational variables used during parsing, see 6.1.1, on page 66. Moreover, the function `new_muexpr_lined_node` is defined, which takes care of the correct manipulation of the above stack when creating new fixpoint nodes. Finally, the function `new_regexpr_infoft_node` converts the "infinitely often" operator into ω-regular expressions, as shown previously in Section 4.3.

`mucalculusCmd.c`. This source file contains the functions `mucalculus_init` and `mucalculus_quit`, which provide the initialization and de-initialization, respectively, of the whole `mucalculus` module. Moreover, this module provides the "check_muspec" command for model-checking μ-calculus specifications (`mucalculus_check_muspec`).

6.3 An example of model in NuSMV

In this section we present a complete example of specification in NuSMV, with properties expressed both in μ-calculus and in ω-**CTL**. Actual code is in figure 6.3.

This is the same example previously shown in Chapter 3 (figure 3.5). However, now we consider fewer properties, shown below.

- Mutual exclusion. This property is expressed directly with the following μ-calculus formula

  $$\nu\, x.\ \neg(s_0 = \mathit{critical} \wedge s_1 = \mathit{critical}) \wedge \mathbf{AX}\ x\ , \tag{6.7}$$

  which is equivalent to the **CTL** formula

  $$\mathbf{AG}\neg(s_0 = \mathit{critical} \wedge s_1 = \mathit{critical})\ . \tag{6.8}$$

  In (our extension of) NuSMV, the previous formula 6.7 is written as:

  ```
  SPEC
    NU x !(s0 = critical & s1 = critical) & AX RELVAR x
  ```

  Note that we have to syntactically distinguish relational variables from other subformulae, hence we prefix occurrences of relational variables with the keyword `RELVAR`.

- No starvation. We express this property with the ω-**CTL** formula:

  $$\mathbf{AG}(s_0 = \mathit{trying} \Rightarrow$$
  $$\mathbf{AF}(\{pr_0.\mathit{running}, pr_1.\mathit{running}, \neg(s_1 = \mathit{critical})\}^{\mathrm{i.o.}}, s_0 = \mathit{critical}))\ ,$$

  which in the NuSMV concrete syntax is rendered as

  ```
  MUSPEC
   AG (s0 = trying ->
    AF [ {[pr0.running], [pr1.running],
    [!(s1 = critical)]} INF, s0 = critical ])
  ```

  Note that, prior to being actually model-checked, the above specification has to be translated into μ-calculus (done internally).

In the following part we will show experimental results and point out ideas for further research.

```
MODULE main

VAR
s0: {noncritical, trying, critical};
s1: {noncritical, trying, critical};
turn: boolean;

pr0: process prc(s0, s1, turn, 0);
pr1: process prc(s1, s0, turn, 1);

ASSIGN init(turn) := 0;

MUSPEC NU x !(s0 = critical & s1 = critical)
  & AX RELVAR x

MUSPEC AG (s0 = trying ->
  (AF [ {[pr0.running], [pr1.running],
  [!(s1 = critical)]} INF, s0 = critical ]))

MODULE prc(state0, state1, turn, turn0)

ASSIGN init(state0) := noncritical;

next(state0) := case
   (state0 = noncritical) : {trying,noncritical};
   (state0 = trying) & (state1 = noncritical): critical;
   (state0 = trying) & (state1 = trying) &
                               (turn = turn0): critical;
   (state0 = critical) : {critical,noncritical};
   1: state0;
esac;

next(turn) := case
   turn = turn0 & state0 = critical: !turn;
   1: turn;
esac;

--Not needed! Fairness is in the temporal specification!
--FAIRNESS running
--FAIRNESS !(state0 = critical)
```

Figure 6.3: Simple mutual exclusion algorithm in NuSMV, with μ-calculus and ω-**CTL** specifications.

Part III

Experimental evaluation

Chapter 7

Empirical results and comparison

In this chapter we present several examples where we apply both:

- Our μ-calculus algorithm (implicitly through the translation from ω-**CTL** to μ-calculus), and
- The logic ω-**CTL**, in particular when we need fairness constraints.

Most examples are part of the NuSMV distribution, while others are taken from the available literature. Moreover, we develop and present various versions of the Peterson's algorithm for ensuring mutual exclusion and study them in detail.

The rest of the chapter is organized as follows. In Section 7.1 we briefly describe the evaluation methodology. In Section 7.2 we present examples which do not make use of fairness, while in Section 7.3 we consider examples with fairness. In particular, in Section 7.3.1 we consider several mutual exclusion protocols, paying particular attention to Peterson's algorithm (Section 7.3.2).

7.1 Evaluation methodology

In this section we present the evaluation and comparison methodology that will be used throughout the chapter. We compare our and NuSMV model-checking algorithms with respect to the number of iterations until convergence and, when possible, the elapsed time. We will consider several models in the next sections and, for each model, several properties. Each property has has been numbered with a progressive index, which runs within the model, i.e., it restarts when passing from a model to another. Results are presented in this general format:

Model name, Property index, Pattern, True/False, **CTL** iter's, μ-calculus iter's

For example the line

Model	#	Property	**CTL**	μ-calculus
`guidance`	1	$\mathbf{AG}\ (\star \Rightarrow \mathbf{AF}\ \star)$	29	29

is to be read as: The first property of the `guidance` model has the form $\mathbf{AG}\ (\star \Rightarrow \mathbf{AF}\ \star)$ (where $\star$ is a generic place-holder for atomic formulae) and it took 29 iterations to reach convergence both in the NuSMV **CTL** and in our μ-calculus model-checking algorithm. In particular, the formula we actually verified was

```
AG(!cg.idle -> AF(cg.finished))
```

where the atomic formulae `cg.idle` and `cg.finished` are relative to the `guidance` model and will not be discussed here.

Please note that ω-**CTL** formulae are converted into μ-calculus formulae before model-checking, hence the actual model-checking is done on μ-calculus formulae. So, we refer to μ-calculus formulae and not to ω-**CTL** formulae in this chapter. However, the following experiments do not only measure the performance of our μ-calculus model-checker (from Chapter 5), but the overall "end-to-end" approach is considered, including the effectiveness of the translation from ω-**CTL** to μ-calculus (presented in Chapter 4).

All the following tests were performed on an Intel Pentium IV 2.8 GHz, 1Gb RAM, 1Mb cache L2.

7.2 Models without fairness

In this section we compare the performance of our μ-calculus model-checking algorithm against NuSMV's one on **CTL** formulae. No fairness constraint is considered here. We studied seven examples, taken from the NuSMV distribution. The aim here is to verify that our μ-calculus model-checking algorithm is no worse than NuSMV on common **CTL** formulae (without involving fairness). Refer to table 7.1 for empirical results (the format of this table has been previously described in Section 7.1).

We can draw some conclusion from the above table. In general, we can see that the two algorithms perform almost identically when fairness is not employed. Sometimes our μ-calculus model-checking algorithm took one more iteration than NuSMV's, but this is related to the slightly different way of counting iterations in the two algorithms. Hence, if the number of iterations differs by only one unit, then we can say that, from a practical point of view, they take the same number of iterations. However, there are cases where our algorithm performed noticeably better, namely, with properties where $\mathbf{A}(\star\,\mathbf{U}\,\star)$ is involved. The reason is as follow: while NuSMV treats $\mathbf{A}(f\,\mathbf{U}\,g)$ as a shorthand for $\neg\mathbf{E}(\neg g\,\mathbf{U}\,\neg f \wedge \neg g) \wedge \neg\mathbf{EG}\neg g$, our approach translates **AU** directly in the fixpoint representation

$$\mu X.\, g \vee (f \wedge \mathbf{AX}X)\ ,$$

which apparently results into a more efficient verification.

Moreover, we should note that in preliminary experiments the running time of our algorithm was not as good as NuSMV, although iterations were quite the same. This happened especially on the `msi_wtrans` model. Then, we discovered that NuSMV model-checking code used the "incremental" representation of approximants which we described in Section 6.1.2, and that was the cause of NuSMV higher performance. Next step was to adopt this enhancement in our μ-calculus algorithm. But problems arose, as the incremental representation was specifically studied for **CTL** formulae (in particular, for the **EU** modality) and it was no at all obvious if it could be lifted to the whole μ-calculus. So, as shown in theorems 2.2 and 2.3, and in Section 6.1.2, we found general conditions under which the incremental representation is correct when applied to μ-calculus formulae, and we embedded it in our algorithm. Fortunately, μ-calculus formulae resulting from the translation from **CTL** were all amenable to this simplification, and we succeeded in matching NuSMV performance.

7.3 Models with fairness

In this section we present the results we obtained on several NuSMV models which make use of fairness constraints.

In this section we consider 8 general examples, which are shown in tables 7.2 and 7.3. The presentation format has been slightly altered to express dependence on fairness. We compare **CTL** formulae with fairness constraints against equivalent μ-calculus formulae, *obtained by translation from suitable ω-**CTL** specifications*. However, such specifications are globally indicated with the label"μ-calculus", as already pointed out in Section 7.1.
Moreover, in some cases we were able to prove properties under fewer fairness assumptions and we compare the performance of the original property against this enhanced version. Consider, for example, the `reactor` model:

Model	#	Property pattern	T/F	**CTL** (iter's)	μ-calculus (iter's)	Fairness constr's
`reactor`	1	**AG AF** $\star$	+	49	27	2 of 2
					45	1 of 2
					21	1 of 2
					23	0 of 2

The first line compares a **CTL** formula of the form **AG AF** $\star$, with full fairness constraints (actually, there are 2 fairness constraints), against an equivalent μ-calculus formula (not shown). In the example, 49 iterations are needed for **CTL** and 27 for μ-calculus. Then, the μ-calculus property was modified in order to take into account fewer fairness constraints, providing that the required property remains true. When taking into account only the first constraint, our algorithm needs 45 iterations to reach convergence, while 21 with only the second constraint and 23 without any constraint. (This is interesting, since actually

Model	#	Property pattern	**CTL** (iter's)	μ-calculus (iter's)
`counter`	1	$\mathbf{AG\ AF}\ \star$	10	10
`dme2`	1	$\mathbf{AF}\ \star$	1	2
`gigamax`	1,2	$\mathbf{AG\ EF}\ \star$	9	10
	3	$\mathbf{AG}\ \star$	1	2
`guidance`	1	$\mathbf{AG}\ (\star \Rightarrow \mathbf{AF}\star)$	29	29
	2	$\mathbf{AGEF}(\star \wedge \mathbf{EF}\star)$	33	34
	3,6,13-16, 18,19,22	$\mathbf{AG}\ \star$	1	2
	4	$\mathbf{AG}\ (\star \Rightarrow \mathbf{E}(\star\,\mathbf{U}\,\star))$	3	3
	5,7,8	$\mathbf{AG}\ (\star \Rightarrow \mathbf{A}(\star\,\mathbf{U}\,\star))$	23	13
	9	$\mathbf{AG}\ (\star \Rightarrow \mathbf{AXAU})$	16	5
	10	$\mathbf{AG}\ (\star \Rightarrow \mathbf{A}(\star\,\mathbf{U}\,\star))$	6	3
	11	$\mathbf{AG}\ (\star \Rightarrow \mathbf{AG}\star)$	16	16
	12	$\mathbf{AG}\ (\star \Rightarrow \mathbf{A}(\star\,\mathbf{U}\,\star))$	41	29
	17	$\mathbf{AG}(\Rightarrow \mathbf{AG}(\Rightarrow \mathbf{AG}))$	40	40
	20,23,24	$\neg\mathbf{E}(\star\,\mathbf{U}\,\star)$	2	2
	21	$\mathbf{AG}\ (\star \Rightarrow \mathbf{E}(\star\,\mathbf{U}\,\star))$	32	33
	25,27	$\mathbf{AG}\ (\star \Rightarrow \mathbf{AG}\star)$	28	28
	26	$\mathbf{AG}\ (\star \Rightarrow \mathbf{AG}\star)$	23	23
`msi_wtrans`	1	$\mathbf{AG}\ (\mathbf{EF} \wedge \mathbf{EF} \wedge \mathbf{EF})$	44 (24 s)	45 (23 s)
	2	$\mathbf{AG\ EF}\ \star$	14 (5 s)	15 (6 s)
	3	$\mathbf{AG\ EF}\ \star$	16 (9 s)	17 (9 s)
	4	$\mathbf{AG\ EF}\ \star$	16 (6 s)	17 (5 s)
	5,6,13,19	$\mathbf{AG}\ \star$	1	2
	7	$\mathbf{AG\ EF}\ \star$	16 (16 s)	17 (16 s)
	8	$\mathbf{AG\ EF}\ \star$	16 (16 s)	17 (16 s)
	9	$\mathbf{AG\ EF}\ \star$	13 (27 s)	14 (25 s)
	10	$\mathbf{AG}\ \star$	8 (3 s)	8 (4 s)
	11	$\mathbf{AG\ EF}\ \star$	19 (8 s)	20 (8 s)
	12	$\mathbf{AG\ EF}\ \star$	16 (13 s)	17 (13 s)
	14,15,16	$\mathbf{AG\ EF}\ \star$	5 (1 s)	6 (1 s)
	17	$\mathbf{AG}\ (\star \Rightarrow \mathbf{EF}\star)$	22 (8 s)	22 (8 s)
	18	$\mathbf{AG}\ (\star \Rightarrow \mathbf{EF}\star)$	27 (10 s)	27 (11 s)
`prod_cell`	1	$\bigwedge_{i=1}^{15} \mathbf{AG}\ (\star \Rightarrow \mathbf{AF}\star)$	474	474
`robot`	1,6,7	$\mathbf{AG}\ \star$	1	2
	2 (-)	$\mathbf{AG}\ \star$	201	201
	3	$\mathbf{AG}\ (\star \Rightarrow \mathbf{AF}\star)$	52	52
	4 (-)	$\mathbf{AG}\ (\star \Rightarrow \mathbf{AG}\star)$	92	92
	5 (-)	$\mathbf{AG}\ (\star \Rightarrow \mathbf{AF}\star)$	602	602

# = property index, unit measure = number of iterations, s = seconds.
In model `robot`, “2 (-)” (for example) means that property #2 is false.
Formulae of the same form are grouped together, e.g., “14,15,16”. See Sec. 7.1.

Table 7.1: Comparison of NuSMV **CTL** model-checking algorithm and our μ-calculus model-checking algorithm. No fairness assumptions.

the property did not need any fairness assumption!). Please note that this fine tuning of fairness would be impossible to do in NuSMV, since in NuSMV fairness is “global” and the same fairness constraints are common to all properties. Hence in NuSMV we cannot delete fairness constraints only for a property without affecting other properties. Other entries of the table can be read in the same manner.

We can draw some conclusions from tables 7.2 and 7.3. In general, it is fair to say that our method is not worse than NuSMV. Moreover, in many cases we were able to use fewer fairness constraints and to obtain fewer iterations. Two cases deserve special attention:

1. The `reactor` model (table 7.2) exhibits an huge number of iterations when restricting to a particular fairness constraint (the second line of every property). This is an evidence for the fact that fewer fairness constraints do not always result into faster convergence. On the contrary, using fewer fairness constraints may considerably slower convergence.

2. In the second property of the `inverter` model (table 7.3), which is of the form
$$\mathbf{AG}\ (\mathbf{AU} \wedge \mathbf{AU})\ , \tag{7.1}$$
our method needs approximatively twice as much iterations as NuSMV. This happens because in the two subformulae **AU**’s we have to *duplicate* fairness constraints, while NuSMV does not. Indeed, this is one case in which global fairness is an advantage over our more fine-grained approach.

Unfortunately, the models are not large enough to enable the measurement of running times. However, in Section 7.3.2 we will present running times for a family of models with fairness.

7.3.1 Mutual exclusion protocols

In this section we continue the study of models that require fairness assumptions. In particular, we take into consideration various *mutual exclusion* protocols, whose NuSMV models are from [PB03] (which is based on [Kes96]). Briefly, a mutual exclusion protocol (or algorithm) is a method for ensuring that a set of processes have access their own critical region in an exclusive manner. Assume that there are n concurrently executed processes, $P_1, \ldots, P_n$. Each process P_i, $i \in [0, n]$, has the general structure

...
ncs_i: Noncritical section
...
try_i: Trying to access the critical section
...
cs_i: Critical section
...

Model	#	Property pattern	**CTL** (iter's)	μ-calculus (iter's)	Fairness constr's
`reactor`	1	**AG AF** $\star$	49	27	2 of 2
				45	1 of 2
				21	1 of 2
				23	0 of 2
	2	**AG AF** $\star$	104	7	2 of 2
				5	1 of 2
	3,4,6,8, 10,11,13	$\neg$**EF** $\star$	1	2	2 of 2
	5	**AG AF** $\star$	18	17	2 of 2
				606	1 of 2
				5	1 of 2
				65	0 of 2
	7	$\neg$**EG EG** $\star$	22	17	2 of 2
				328	1 of 2
				5	1 of 2
				40	0 of 2
	9	$\neg$**EG EG** $\star$	9	8	2 of 2
				17	1 of 2
				5	1 of 2
				7	0 of 2
	12	$\neg$**EG EG** $\star$	22	21	2 of 2
				252	1 of 2
				5	1 of 2
				29	0 of 2
	14	$\neg$**EG EG** $\star$	9	8	2 of 2
				59	1 of 2
				5	1 of 2
				21	0 of 2

# = property index.
Unit of measure = number of iterations.
Formulae of the same form are grouped togheter, e.g., "3,4,6,8,10,11,13".
The last column indicates "how much" fairness constraints are used (not shown), e.g., "1 of 2" means that 1 out of 2 constraints are used.

Table 7.2: Comparison of NuSMV **CTL** model-checking algorithm and our μ-calculus model-checking algorithm with fairness assumptions. Part I.

Model	#	Property pattern	T/F	**CTL** (iter's)	μ-calculus (iter's)	Fairness constr's
`abp4`	1	**AG AF** $\star$	+	93	46	6 of 6
	2	**AG AF** $\star$	+	93	56	6 of 6
`inverter`	1	**AG AF** $\wedge$ **AF**	+	65	37	3 of 3
					27	2 of 3
	2	**AG AU** $\wedge$ **AU**	+	67	117	3 of 3
					75	2 of 3
	3	**AG** $\star$	-	5	5	3 of 3
	4,5	**AG** $\star$	+	1	1	3 of 3
	6	**AG AF** $\star$	+	23	13	3 of 3
`lift`	1-4	**AG** $\star$	+	1	2	1 of 1
	5	**AG** $\star \Rightarrow$ **AF** $\star$	+	55	51	1 of 1
	6	**AG** $\star \Rightarrow$ **AF** $\star$	+	31	31	1 of 1
`prod_cons`	1	**AG** $\star \Rightarrow$ **AF** $\star$	+	103	91	3 of 3
	2	**AG** $\star \Rightarrow$ **AF** $\star$	-	359	261	3 of 3
	3	**AG** $\star \Rightarrow$ **AF** $\star$	-	358	264	3 of 3
	4	**AG** $\star \Rightarrow$ **AF** $\star$	-	385	291	3 of 3
`ring`	1	**AG AF** $\wedge$ **AF**	+	89	42	1 of 1
`semaphore`	1	**AG** $\star$	+	1	2	3 of 3
	2	**AG** $\star \Rightarrow$ **AF** $\star$	+	34	21	3 of 3
					5	1 of 3
`token2`	1	**AGEFAG** $\star$	+	17	17	2 of 2
	2	$\neg$**EFAG** $\star$	+	10	10	2 of 2
	3	**EFEUEU**	+	25	25	2 of 2
	4	**AGEFAGAU**	+	2870	2993	2 of 2

# = prop.ty index, T/F = whether (+) or not (-) the property is true.
Unit of measure = number of iterations.
Formulae of the same form are grouped togheter, e.g., "4,5".
The last column indicates "how much" fairness constraints are used (not shown), e.g., "1 of 2" means that 1 out of 2 constraints are used.

Table 7.3: Comparison of NuSMV **CTL** model-checking algorithm and our μ-calculus model-checking algorithm with fairness assumptions. Part II.

We assume that each process remains only a finite amount of time in both the noncritical region and the critical region. However, we *do not* assume that each process remains a finite amount of time in the trying-to-access region, indeed this is a property of the protocol we wish to prove.

Among all, we are interested in the following properties of mutual exclusion protocols:

- *Mutual exclusion.* If there are n process, it is not the case that any two of them are simultaneously in their own critical section. In **CTL** this can be expressed as
$$\mathbf{AG} \bigwedge_{1\leq i<j\leq n} \neg(cs_i \wedge cs_j)\ . \tag{7.2}$$
- *Non-blocking.* It is not the case that a process remains forever in its noncritical section. In **CTL**, this is expressed as (for each processes P_i)
$$\mathbf{AG}\ (ncs_i \Rightarrow \mathbf{AF}\ try_i)\ . \tag{7.3}$$
- *No starvation.* If a process P_i wants to enter its critical section, then it will eventually succeed. In **CTL**:
$$\mathbf{AG}\ (try_i \Rightarrow \mathbf{AF}\ cs_i)\ . \tag{7.4}$$
- *Bounded overtaking.* If process P_i wants to enter its critical section, then it is not the case that there exists another process P_j which enters its own critical section twice, while P_i still is awaiting for entering. For two processes P_1, P_2, with $i = 1, j = 2$, bounded overtaking can be expressed in **CTL** as
$$\begin{aligned} \mathbf{AG}\ (try_1 \quad \Rightarrow \quad & \mathbf{A}(\neg cs_2\, \mathbf{U}\, cs_1) \vee & (7.5)\\ & \mathbf{A}(\neg cs_2\, \mathbf{U}\, cs_2 \wedge \mathbf{A}(cs_2\, \mathbf{U} \neg cs_2 \wedge \mathbf{A}(\neg cs_2\, \mathbf{U}\, cs_1)))).& (7.6)\end{aligned}$$
However, we will use a slightly different **CTL** formula, which is more simple, but it is enough for our purposes:
$$\mathbf{AG}\ (try_1 \wedge cs_2 \Rightarrow \mathbf{A}(cs_2\, \mathbf{U}\, (\neg cs_2 \wedge \mathbf{A}(\neg cs_2\, \mathbf{U}\, cs_1))))\ . \tag{7.7}$$
(Note that the last equation does not exactly express bounded overtaking in the form we presented it.)

We consider five mutual exclusion protocols, as shown in table 7.4. Each model consists of two concurrently running processes, P_1, P_2. This time we do not need to show the property patterns as we did before, as they are clear from the previous description. As before, we first compare a **CTL** property versus an equivalent μ-calculus formula, under the *same* fairness assumptions, which are:

1. Process P_1 is executed infinitely often.

2. Process P_1 does not remain forever in its critical section.
3. Process P_1 does not remain forever in its noncritical section.

Analogous assumptions are made for P_2, with a total of 6 fairness constraints.

Subsequently, we proceed relaxing fairness on our formulae (taking the point of view of process P_1). Relaxation is different for each property and it is described below:

- No starvation. We require only that P_1 and P_2 are executed infinitely often and that P_2 does not remain forever in its critical section. So we really need only 3 fairness assumptions to prove no starvation for P_1 (instead of 6):

  $$1)\ \mathbf{AG}\ (try_1 \Rightarrow \mathbf{AF}(\{run_1, run_2, \neg cs_2\}^{\text{i.o.}}, cs_1)\ .$$

- Non-blocking. This time, we relax fairness in two consecutive steps:
  - In the first step, we only need that P_1 is executed infinitely often and does not remain in its noncritical region forever. 2 fairness constraints here:

    $$1)\ \mathbf{AG}\ (ncs_1 \Rightarrow \mathbf{AF}(\{run_1, \neg ncs_1\}^{\text{i.o.}}, try_1)\ .$$

  - We can do better, requiring only that P_1 does not remain in its noncritical region forever. (This implies that P_1 is executed infinitely often as long as P_1 is in the noncritical region). Note that this is quite obvious in all but one case. 1 fairness constraint:

    $$2)\ \mathbf{AG}\ (ncs_1 \Rightarrow \mathbf{AF}(\{\neg ncs_1\}^{\text{i.o.}}, try_1)\ .$$

- Bounded overtaking. This time we use two distinct sets of fairness constraints on different part of the formula:

  $$\begin{aligned}1)\ \mathbf{AG}\ (try_1 \wedge cs_2 \quad \Rightarrow \quad & \mathbf{AU}(\{\neg cs_2\}^{\text{i.o.}}, cs_2,\\ & \neg cs_2 \ \wedge\ \mathbf{AU}(\{run_1, run_2\}^{\text{i.o.}}, \neg cs_2, cs_1)))\ .\end{aligned}$$

  In fact, if P_2 is in its critical section, in order to prove that P_2 will eventually leave the critical section we only need that $\neg cs_2$ occurs infinitely often. Analogously, in the inner **AU** we need only that both processes are executed infinitely often in order to prove that P_1 will eventually enter its critical region. This allows to simplify the formula, which results in computational improvements.

These results are summarized in table 7.4. For each model, there are three entries. The first entry compares properties on the basis of the same fairness assumptions. The second and third entries, beginning with "1)" and "2)", respectively, represent the relaxation on fairness discussed above.

Input	Mutual excl.			No starv.			Non-block.			Boun'd overt.		
model		ctl	μtl		ctl	μtl		ctl	μtl		ctl	μtl
`dekker`	+	1	2	+	311	195	+	113	56	-	535	660
1)						124			13			167
2)									7			
`dijkstra`	+	1	2	-	96	99	-	82	85	-	154	534
1)						55			44			125
2)									22			
`hyman`	-	12	12	-	108	113	+	85	60	-	173	568
1)						61			21			138
2)									7			
`knuth`	+	1	2	+	350	269	+	303	208	+	587	782
1)						174			17			179
2)									7			
`peterson`	+	1	2	+	158	84	+	89	43	+	283	526
1)						45			16			122
2)									21			

The + or - indicates whether (+) or not (-) the property is true.
Unit of measure = number of iterations. μtl = μ-calculus temporal logic.
Rows marked with 1) or 2) refer to the properties modified with (progressively) fewer fairness constraints (discussed in the text).

Table 7.4: Comparison of NuSMV **CTL** model-checking algorithm and our μ-calculus model-checking algorithm for various mutual exclusion protocols. Unit measure: number of iterations.

We can make some observations on table 7.4. Except for bounded overtaking, our approach has remarkable benefits. The number of iterations is in general smaller and when we reduce fairness (the lines marked with "1)" and "2)"), improvements are even more manifest. The last observation holds also for bounded overtaking. However, when proving bounded overtaking with full fairness assumptions, our approach is considerably worse. This is for the same reason as in the previous section. The reason is that in the bounded overtaking formula there are two occurrence of the **AU** formula, and each occurrence has its set of (full) fairness constraints. And, obviously, the *duplication* of fairness constraints leads to slower convergence. However, note that when we reduce assumptions on bounded overtaking, we always result in better performance.

7.3.2 Peterson's algorithm

In this section we analyze Peterson's algorithm for mutual exclusion. (Peterson's algorithm first appeared in [Pet81], while our presentation is from [Pet05].) We present the algorithm in general for n processes and we verify it for at most 5 processes. Moreover, we consider two versions of the algorithm, depending on whether certain tests are atomic or not. Unlike what done in the previous section, here we do *not* require that each process remains a finite amount of time in the noncritical section.

Assume that there are n processes. The general form of process P_i is shown in algorithm 7.1. All elementary operations are assumed to be atomic. Elementary operations are:

- The assignments $Q[i] := j$, $S[j] := i$ and $Q[i] := 0$, and
- Each of the tests $Q[k] < j$, $S[j] \neq i$.

Now consider the **wait**() statement on line ask_i. It is not clear if it has to been executed atomically or not. Hence, we should consider both cases:

1. *Non-atomic version.* In this case, we must evaluate the $n-1$ conditions on Q and the condition on S sequentially. There are a lot of ways of sequencing these conditions. We just chose one. Hence, instead of

$$\mathbf{wait}((\forall k \neq i.\ Q[k] < j) \vee (S[j] \neq i))$$

   we have

```
test $k_1$      **if** $Q[k_1] < j$ **then goto** test $k_2$ **else goto** test $s$;
test $k_2$      **if** $Q[k_2] < j$ **then goto** test $k_3$ **else goto** test $s$;
test $k_3$      ...
test $k_{n-1}$  **if** $Q[k_{n-1}] < j$ **then goto** $cs_i$ **else goto** test $s$;
test $s$        **if** $S[j] \neq i$ **then goto** $cs_i$ **else goto** test $k_1$;
```

The $k_1, \ldots, k_{n-1}$ form a sequence of $n-1$ distinct numbers, each in $[1, n]$ and different from i, satisfying

$$k_1 \leq k_2 \leq \ldots \leq k_{n-1} \; .$$

(It is easy to see that given i and n, all the $k_1, \ldots, k_{n-1}$ are uniquely determined.)

2. *Atomic version.* In this case, each time the **wait**() statement is executed, all the tests are performed simultaneously and in only one execution step.

We wrote a Prolog program that, given n and a flag (whether the version is atomic or not), returns the corresponding NuSMV model. This is quite effective, since for $n > 3$ the model is quite large and hand-coding it would have been error-prone.

In table 7.5 we present the results with Peterson's algorithm. We always prove properties with respect to process P_1. We use the following two sets of fairness constraints (for n processes):

a) All processes are scheduled infinitely often and neither remains in its critical section forever. Using the $\{\cdot\}^{\text{i.o.}}$ notation, this is equivalent to

$$\{\texttt{running}_1, \ldots, \texttt{running}_n, \neg cs_1, \neg cs_2, \ldots, \neg cs_n\}^{\text{i.o.}} \; . \tag{7.8}$$

b) All processes are scheduled infinitely often and neither but P_1 remains in its critical section forever:

$$\{\texttt{running}_1, \ldots, \texttt{running}_n, \neg cs_2, \ldots, \neg cs_n\}^{\text{i.o.}} \; . \tag{7.9}$$

From table 7.5 we can see that our handling of fairness always allows for faster convergence. Also, when the size of the models is sufficiently large for elapsed-time being relevant (i.e., for the cases 4, 4A and 5A) we can see that our approach (almost) always results in shorter execution times. Moreover, when we use the reduced fairness b), instead of a), we do obtain further improvements.

In the next chapter we will gather together the results we have shown here and we will draw some conclusions.

Algorithm 7.1: Process P_i in the Peterson's algorithm for n processes, $i = 1, \ldots, n$.

```
        while true do
$ncs_i$     non critical section $i$;
            for $j = 1, \ldots, n-1$ do
                $Q[i] := j$;
                $S[j] := i$;
$ask_i$         wait($(\forall k \neq i.\ Q[k] < j) \vee (S[j] \neq i)$);
$cs_i$      critical section $i$;
            $Q[i] := 0$;
```

# proc's (A=atom.)	Mutual exclusion CTL		Mutual exclusion μ-calculus		No starvation CTL		No starvation μ-calculus		Fair-ness.
	N	T	N	T	N	T	N	T	
2	1	N/A	2	N/A	199	N/A	137	N/A	a)
							119	N/A	b)
3	1	N/A	2	N/A	1143	N/A	815	N/A	a)
							757	N/A	b)
4	1	171s	2	8s	4669	140m	3257	151m	a)
							3115	138m	b)
2A	1	N/A	2	N/A	148	N/A	86	N/A	a)
							74	N/A	b)
3A	1	N/A	2	N/A	666	N/A	349	N/A	a)
							319	N/A	b)
4A	1	2s	2	2s	2130	12s	1039	9s	a)
							977	8s	b)
5A	1	320s	2	49s	5389	170m	2636	142m	a)
							2522	131m	b)

N = number of iterations, T = elapsed time (m = minutes, s = seconds).

Table 7.5: Comparison of NuSMV **CTL** model-checking algorithm and our μ-calculus model-checking algorithm for various versions of Peterson's algorithm.

Chapter 8

Conclusions and future research

In this chapter we summarize what we have done, we give link to related work in literature and we present some conclusions and hints for further investigations. The rest of this chapter is organized as follows. In Section 8.1 we present work in literature that is related to and/or inspired both our logic ω-**CTL** and our μ-calculus model-checking algorithm. In Section 8.2 we draw some conclusions from this work and we point out topics that we believe to be worthy of further study.

8.1 Related work

In this section we relate our work to the literature. We give links to similar/related work and we explain in what respect our approach is different.

8.1.1 Work related to ω-CTL

The idea of bringing regular modalities from dynamic logics, like PDL [FL79] and ΔPDL [Str82], to temporal logics is not new. For example, in [VW84] the authors consider extending temporal logic with modalities defined by non-deterministic finite state automata. In particular, they discuss the extension of **CTL*** (that they called **ECTL***) where path formulae are not ordinary **LTL** expressions, but are represented by automata over infinite words. It can be shown that **ECTL*** is more expressive that **CTL***[1]. See also [VW94] (from the same authors), where other kinds of automata are taken into consideration.

[1]This result follows from the following facts. **LTL** has the same expressive power as star-free ω-regular expressions, while non-deterministic Büchi automata achieve full ω-regularity. Hence, if we replace the linear time fragment of **CTL***, i.e., **LTL**, with ω-automata, then we can increase **CTL*** expressiveness.

Along the same line of mixing temporal logic and language-theoretic means, in [EJS93] ω-regular expressions are used instead of automata[2]. In [EJS01] the same authors also give a translation from the dialect of **ECTL*** which considers ω-expressions instead of automata into (a fragment of) μ-calculus. Part of our translation from ω-**CTL** to μ-calculus closely resembles the one in [EJS01]. However, they consider only existentially quantified formulae, since universally quantified formulae require the complementation of ω regular expressions, which is related to the complementation of Büchi automata, a notoriously hard problem [Var07c]. Our approach is different, since we consider *relativized* **CTL** path quantifiers, and not absolute path quantifiers. Consider the ω-**CTL** formula $\mathbf{EG}(a, f)$, for a ω-regular expression a and a formula f. $\mathbf{EG}(a, f)$ is equivalent to the **CTL***-like formula (considering a as a path formula)

$$\mathbf{E}\ (a \wedge \mathbf{G}\, f)\ ,$$

whose complement is

$$\mathbf{A}\ (a \Rightarrow \mathbf{F}\, f)\ ,$$

which is equivalent (by definition) to the ω-**CTL** formula $\mathbf{AF}(a, f)$. Hence, we can see that our semantic does not need the complementation of a, which considerably eases the model-checking problem.

The idea of *relativizing* path quantifiers to a certain set of paths can be found in [EL85, EL86], where an extensive study of various notions of fairness is done. From an high-level point of view, our idea is very similar. However, the details are quite different. In [EL85] fairness is expressed using formulae from a subset of **CTL***, and an efficient model-checking algorithm is given for a (further restricted) class of constraints in a certain normal form, which seems to be enough for practical purposes. Instead, our approach is a bit more general, since path quantifiers are interpreted over paths defined by *arbitrary* ω-regular expression, not necessarily restricted to express fairness. However, at present time the two approaches are incomparable, since ω-**CTL** cannot express strong fairness, as we do not have intersection for ω-regular expressions. This is also discussed in Section 8.2 below.

From the point of view of practical usage, the machinery of ω-automata seems not amenable to the practitioner. Hence, formalisms have been developed to improve temporal logic usability and to close the gap between the academic world of researchers and the industry world of engineers. Several studies found that engineers felt comfortable with (classical) regular expressions. Hence, formalisms for mixing temporal logics with regular modalities were developed. The branching-time temporal logic Sugar [BBDE$^+$01] focuses on usability. On the one hand, Sugar adds the power of regular expressions to **CTL**, increasing its expressiveness and readability. On the other hand, Sugar adds syntactic sugaring (hence the name), which does not augment the expressive power of the logic, but allows complex properties to be expressed more succinctly. Sugar is

[2][EJS93] was also the first paper where the model-checking problem for the μ-calculus was shown to be in NP∩coNP.

used inside the RuleBase model-checker at IBM. We argue that we could easily extend ω-**CTL** with the new constructs from Sugar, with no added complexity to the translation into μ-calculus.

The linear-time correspondent of Sugar is RELTL [BFG$^+$05]. While the logic **ECTL*** is **CTL***+ω-automata, RELTL can be seen as **LTL**+regular-expressions, i.e., RELTL is the result of adding a regular expressions over state predicates to **LTL**. Moreover, this suffices for achieving ω-regularity. The industrial version of RELTL is ForSpec from Intel [AFF$^+$02], which added a lot of syntactic sugaring (e.g., clocks and resets). The main features of ForSpec were finally adopted in the industry-standard IEEE 1850 PSL [FMW05], which was also based on Sugar.

Also the (alternation-free) μ-calculus has been extended with regular PDL-like modalities. In [MS03] the authors enrich the action-based μ-calculus with ACTL-like formulae and PDL-like regular expressions (although with some syntactic restrictions). This new specification language is the basis of the EVALUATOR 3.0 model-checker. Also in this work, expressive power is traded for user-friendliness.

8.1.2 Work on model-checking algorithms for μ-calculus and its fragments

The first asymptotic improvement for global, symbolic model-checking came from Emerson and Lei [EL86], where it was shown that successively nested fixpoints of the same type do not increase the complexity of the computation. Emerson and Lei introduced the notion of *alternation depth* d of a μ-calculus formula, as a measure of the number of genuine alternations in fixpoints. Their algorithm had a complexity of $O(n^d)$, where n is the size of the system. Successively, in [CS92] the representation of fixpoint formulae via mutually recursive systems of equations led to complexity improvements for the *alternation-free* fragment of the μ-calculus. Moreover, in [CKS93] the representation via equational systems was lifted to the full μ-calculus, i.e., with alternating fixpoints. However, all these improved algorithms had a worst-case complexity of $O(n^d)$, although with smaller constants.

The major asymptotic improvement came in [BCJ$^+$94], where a new algorithm of worst case complexity roughly equal to $O(n^{d/2})$ was given. After roughly 15 years, this apparently is the best global algorithm for the full μ-calculus (with respect to worst case complexity). However, we are not aware of implementations of the algorithm in [BCJ$^+$94], mainly because 1) also space complexity is $O(n^{d/2})$, which is impractical in many cases, and 2) tangible benefits of using [BCJ$^+$94] can result only for complex μ-calculus formulae of alternation depth ≥ 3, which are rather unlikely to appear in actual examples.

Hence, researchers tried to find specialized, more efficient algorithms for significant fragments of the μ-calculus. We already mentioned the algorithm in [CS92] for the alternation-free μ-calculus. In [EJS01] are identified the two noticeable fragments L1 and L2, for which an efficient procedure based on automata was given. Moreover, in [BC96] a *local* procedure is given for L2.

Until now, we covered only efficient algorithms based on putting restrictions on the logic. Efficient algorithms can also be obtained by restricting the underlying structure. In [Obd03] an algorithm for the full μ-calculus based on parity games is given and it is shown that the model-checking complexity is linear when the *tree-width* of the underlying structure is bounded[3]. In [Obd03] it is also noted that structures resulting from software programs typically have bounded (and small) tree-width, hence the proposed model-checking algorithm could be of practical significance.

8.2 Conclusions and future directions

We have presented a promising approach to fair model-checking. We introduced the novel logic ω-**CTL**, which allowed us to express (weak) fairness assumptions. We gave a translation from ω-**CTL** to μ-calculus. The formulae resulting from this translation can be model-checked with standard μ-calculus model-checking algorithms. Moreover, we also presented a novel μ-calculus model-checking algorithm. Unfortunately, μ-calculus formulae resulting from our translation are all of alternation depth ≤ 2. Hence, the potential benefits of our algorithm were not empirically measured.

Final remarks

From the results discussed in Chapter 7 we argue that our approach offers several advantages. In particular:

- On models without fairness, our handling of the **CTL** modality **AU** is more efficient.
- Incremental representation of relational variables can be lifted to a significative subset of μ-calculus, resulting in faster iterations.
- On models with fairness, fine tuning of fairness constraints for each formula often results in improved performance. Moreover, our approach allows to use different fairness assumptions on different subformulae, which is impossible to do in NuSMV.

However, there are also situations in which the global approach to fairness of NuSMV is preferable:

- On models with fairness, when there are many subformulae that are interpreted with respect to the same fairness assumptions, the global approach of NuSMV is preferable. In that case, our approach is less compact and results in much many iterations.

[3] Roughly speaking, tree-width measures how close is a given graph to being a tree. It is known that μ-calculus model-checking has linear complexity on trees.

- It is not always the case that fewer fairness assumptions imply faster convergence. Hence, the ability of ω-**CTL** to express finer-grained fairness assumptions not necessarily entails fewer iterations until convergence.

Future directions

We can now give some hints for future directions.

- The syntax of ω-regular expressions we presented is rather limited. In particular, we do not take into account neither intersection "$\wedge$" nor negation "$\neg$" of ω-regular expressions. In fact, negation of expressions corresponds to negation of Büchi automata, which is a difficult operation [Var07c]. If we were able to use intersection, we could simplify our presentation of fairness. Moreover, we would also be able to express *strong fairness*. In fact, using intersection of ω-regular expressions, we can express:

  - Weak fairness. Each of a_i's must occur infinitely often:
  $$\bigwedge_i ((\neg a_i)^\star \cdot a_i)^\omega \ . \tag{8.1}$$
  Note that weak fairness is also expressible without intersection, as we have previously showed in Section 4.3. However, the above equation 8.1 shows that weak fairness can be expressed more readably and more compositionally using intersection.
  - Strong fairness. Each of b_i's must occur infinitely often when a_i does:
  $$\bigwedge_i (\mathbf{true}^\star \cdot (\neg a_i)^\omega + ((\neg b_i)^\star \cdot b_i)^\omega) \ . \tag{8.2}$$
  Note that strong fairness *cannot be (concisely) expressed* without intersection.

  From these two observation we argue that if were were able to use intersection in expressions, fairness could be expressed in a more elegant and compositional way.

- On the same generalizing trend as in the previous point, it would be interesting to extend ω-regular expressions, which is a linear-time formalism, into *branching ω-regular expressions*. The relation-ship between branching and linear expressions should be similar to the relationship between **CTL*** and **LTL**. The syntax of branching ω-regular expressions would be something like
  $$a \ ::= \ p \mid a\&a \mid a+a \mid \neg a \mid a \cdot a \mid a^+ \mid a^\omega \mid \exists a \mid \forall a \ , \tag{8.3}$$
  where the quantifiers $\exists$ and $\forall$ express the branching nature of time. Just like **CTL*** can be restricted to **CTL**, also branching ω-regular expressions could be restricted to a proper subset in which the linear operators ("$\star$" is just a place-holder) $\star \cdot \star$, $\star^+$ and $\star^\omega$ strictly alternate with path quantifiers $\exists$ and $\forall$. Hence, an efficient translation in μ-calculus should be given.

- Our μ-calculus model-checking algorithm employs a static strategy of caching previous values. We have seen that the innermost fixpoint variable of a formula of alternation depth d must remember about $d/2$ previous values. This is something in between the $O(n^{d/2})$ of [BCJ+94] and the $O(1)$ of [EL86]. More flexible caching strategies could be developed, based on available memory. For example, innermost fixpoints could remember more values than middle fixpoints. A caching strategy based on available memory could result in the optimal memory/time tradeoff for a given problem-instance/underlying-machine pair.

- When calculating a topmost fixpoint, it is not always necessary to reach convergence. In fact, since we are interested in checking whether the set of initial states I_0 is included in the limit value A^ω, two cases naturally arise:

  - If we are evaluating a *minimal fixpoint*, then for all i $A^i \subseteq A^{i+1}$. Hence, if there exists an i s.t. $I_0 \subseteq A^i$, then the property certainly holds, and we do not need to reach A^ω.
  - If we are evaluating a *maximal fixpoint*, then for all i $A^{i+1} \subseteq A^i$. Hence, if there exists an i s.t. $I_0 \not\subseteq A^i$, then the property is certainly false. Also in this case, there is no need to reach A^ω.

  This reasoning naturally extends to nested fixpoint of the same type. It would be interesting to study whether this *early stop* method can be carried to inner, alternating fixpoints, thus allowing for fewer iterations.

  Another interesting point is to merge this approach with the *incremental representation* method of Section 6.1.2. The idea is that we could always use the incremental representation, also in the cases in which it is not sound. However, if spurious counterexamples are generated (or the specification is verified when it is indeed false), then appropriate recovering methods should be set-up.

- Sometimes model-checking can be done separately for subformulae. It could be useful to determine when the following relations could be applied to model-checking:

$$\begin{aligned}
\mu X.\, \varphi_1(X) \wedge \varphi_2(X) &\subseteq (\mu X.\, \varphi_1(X)) \wedge (\mu X.\, \varphi_2(X)) && (8.4)\\
\nu Y.\, \varphi_1(Y) \vee \varphi_2(Y) &\supseteq (\nu Y.\, \varphi_1(Y)) \vee (\nu Y.\, \varphi_2(Y))\ . && (8.5)
\end{aligned}$$

  In fact, in some cases the evaluation of right-hand sides could be more efficient, although in principle twice as much as iterations could possibly be required (we hope in fewer time per iteration). Also the following two relations could be useful:

$$\begin{aligned}
\mu X.\, \varphi_1(X, \mu Y.\, \varphi_2(X,Y)) &\subseteq \mu X.\, \varphi_1(X, \nu Y.\, \varphi_2(X,Y)) && (8.6)\\
\nu X.\, \varphi_1(X, \nu Y.\, \varphi_2(X,Y)) &\supseteq \nu X.\, \varphi_1(X, \mu Y.\, \varphi_2(X,Y))\ . && (8.7)
\end{aligned}$$

Note that left-hand sides can be evaluated in $O(n)$ steps. Moreover, in the worst case they can nonetheless be utilized as initial approximants in the computation of right-hand sides.

- Finally, practical examples of complex μ-calculus formulae of alternation depth ≥ 3 would be useful for comparing our algorithm with both Emerson and Lei's and Browne's et al. algorithms. Such practical examples could be automatically derived from novel translations of (meaningful) formulae in other temporal logics, like **CTL***, into μ-calculus formulae of high alternation depth.

List of Algorithms

List of Tables

List of Figures

Index

Bibliography

[AFF$^+$02] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 296–211, London, UK, 2002. Springer-Verlag.

[BBDE$^+$01] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic Sugar. *Lecture Notes in Computer Science*, 2102, 2001.

[BC96] G. Bhat and R. Cleaveland. Efficient local model-checking for fragments of the modal mu-calculus. *Lecture Notes in Computer Science*, 1055:107–126, 1996.

[BCJ$^+$94] A. Browne, E. M. Clarke, S. Jha, D. E. Long, and W. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In D. Dill, editor, *Proceedings of CAV '95 Workshop*, pages 338–350. Springer-Verlag, June 1994.

[BCJM96] S. Berezin, E. Clarke, S. Jha, and W. Marrero. Model checking algorithms for the μ-calculus. Technical Report CMU-CS-96-180, Carnegie Mellon University, 1996.

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, 1991. North-Holland.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Information and Computation 98(2)*, pages 142–170, 1992.

[BFG$^+$05] Doron Buston, Alon Flaisher, Orna Grumberg, Orna Kupferman, and Moshe Y. Vardi. Regular vacuity, 2005.

[Bra97] J. C. Bradfield. The modal mu-calculus alternation hierarchy is strict. *Theoretical Computer Science*, 195:133–153, 1997.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers C-35(8)*, pages 677–691, 1986.

[Büc62] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of International Congress on Logic, Method, and Philosophy of Science (1960)*, pages 1–12, Stanford, 1962. Stanford University Press.

[CCG$^+$02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[CCJ$^+$a] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. *NuSMV 2.4 Programmer Manual.*

[CCJ$^+$b] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. *NuSMV 2.4 User Manual.*

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981.* Springer, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions of Programming Languages and Systems*, 8:244–263, 1986.

[CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 2000.

[Cho74] Y. Choueka. Theories of Automata on omega-Tapes: A Simplified Approach. *JCSS*, 8(2):117–141, 1974.

[Chu57] A. Church. Application of recursive arithmetics to the problem of circuit synthesis. *Summaries of Talks Presented at The Summer Institute for Symbolic Logic*, pages 3–50, 1957.

[CKS92] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal μ-calculus. In G. V. Bochmann and D. K. Probst, editors, *Proceedings of CAV '92 Workshop*, volume 663 of *Lecture Notes In Computer Science*. Springer-Verlag, July 1992.

[CKS93] Rance Cleaveland, Marion Klein, and Bernhard Steffen. Faster model checking for the modal mu-calculus. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 410–422, London, UK, 1993. Springer-Verlag.

[CS92] R. Cleaveland and B. Steffen. A linear–time model–checking algorithm for the alternation–free modal mu–calculus. In Kim G. Larsen and Arne Skou, editors, *Proceedings of Computer Aided Verification (CAV '91)*, volume 575, pages 48–58, Berlin, Germany, 1992. Springer.

[CWA$^+$96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. *Theoretical Computer Science*, 126(1):77–96, 1994.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometime" and "Not Never" Revisited - On Branching versus Linear Time. *Journal of the ACM*, 33(1):151–178, 1986.

[EJS93] E. A. Emerson, C. Jutla, and A. P. Sistla. On model checking for fragments of the μ-calculus. In *Proceedings of Fifth International Conference on Computer Aided Verification, Elounda, Greece*, volume 697 of *Lecture Notes In Computer Science*, pages 385–396, June/July 1993.

[EJS01] E. A. Emerson, C. Jutla, and A. P. Sistla. On model checking for μ-calculus and its fragments. *Theoretical Computer Science*, 258:491–522, 2001.

[EL85] E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time strikes back. In *12th ACM Symposium on Pronciples of Programming Languages*, pages 84–96, January 1985.

[EL86] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.

[Eme95] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland Pub. Co., 1995.

[Eme97] E. A. Emerson. Model checking and the μ-calculus. In N. Immerman and P. Kolaitis, editors, *DIMACS Symposium on Descriptive Complexity and Finite Model*, pages 185–214. American Mathematical Society Press, 1997.

[FL79] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *JCSS*, 18:194–211, 1979.

[FMW05] Harry Foster, Erich Marschner, and Yaron Wolfsthal. IEEE 1850 PSL: The Next Generation, 2005.

[FPP05] F. Fioravanti, A. Pettorossi, and M. Proietti. Automatic proofs of protocols via program transformation. In B. Dunin-Keplicz, A. Jankowski, A. Skowron, and M. Szczuka, editors, *Monitoring, Security, and Rescue Techniques in Multiagent Systems, Advances in Soft Computing Series*, pages 99–116. Springer, 2005.

[Gle96] J. Gleick. A bug and a crash - sometimes a bug is more than a nuisance. *New York Times Magazine*, December 1996.

[Gro96] The VIS Group. VIS: A system for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science,*, pages 428–432. Springer, July 1996.

[Kes96] L. Kesteloot. Survey of mutual-exclusion algorithms for multiprocessor operating systems. Available online: `www.teamten.com/lawrence/242.paper/242.paper.html`, 1996.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *TCS*, 27:333–354, 1983.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 2:125–143, 1977.

[Lam80] L. Lamport. "Sometime" is sometimes "Not Never" - On the Temporal Logic of Programs. In *7th ACM Symposium on Prnciples of Programming Languages*, pages 174–185, January 1980.

[Lam83] L. Lamport. What good is temporal logic? *Information Processing 83, R. E. A. Mason, ed.*, pages 657–668, 1983.

[Lar95] F. Laroussinie. About the expressive power of CTL combinators. *Information Processing Letters*, 54(6):343–345, jun 1995.

[Len96] G. Lenzi. A hierarchy theorem for the mu-calculus. In *ICALP*, volume 1099 of *LNCS*, pages 87–109, 1996.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.

[May97] R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems.* PhD thesis, Institut für Informatik der Technischen Universität München, 1997.

[McM92a] K. L. McMillan. The SMV system, November 06 1992.

[McM92b] K. L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, May 1992.

[MS03] Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. In *Sci. Comput. Program.*, volume 46(3), pages 255–281, Amsterdam, The Netherlands, 2003. Elsevier North-Holland, Inc.

[NV07] Sumit Nain and Moshe Y. Vardi. Branching vs. linear time: Semantical perspective. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2007.

[Obd03] J. Obdrzalek. Fast mu-calculus model checking when tree-width is bounded. In *CAV '03*, volume 2725 of *LNCS*, pages 80–92. Springer, 2003.

[Par81] D. Park. Concurrency and automata on infinite sequences. *Lecture Notes In Computer Science*, 104:167–183, 1981.

[PB03] E. Pek and N. Bogunovic. Verification of mutual exclusion algorithms with SMV system. In *EUROCON 2003. Computer as a Tool. The IEEE Region 8*, volume 2, pages 21–25, September 2003.

[Pet81] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3), 1981.

[Pet05] A. Pettorossi. *Elements of Concurrent Programming (Second Edition).* Aracne, 2005.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pri57] A. Prior. *Time and Modality.* Oxford University Press, 1957.

[QS81] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CAESAR. In *Proceedings of the 5th International Symposium on Programming*, pages 337–350, 1981.

[Rab00] A. Rabinovich. Symbolic model checking for μ-calculus requires exponential time. *Theoretical Computer Science*, 243:467–475, 2000.

[RM00] A. Rabinovich and S. Maoz. Why so many temporal logics climb up the trees? In *Mathematical Foundations of Computer Science*, pages 629–639, 2000.

[Sha97] N. Shankar. Machine-assisted verification using theorem proving and model checking. In M. Broy, editor, *Mathematical Methods in Program Development*. Springer-Verlag, 1997.

[Str82] R. Street. Propositional dynamic logic of looping and converse. *Information and Control*, 54:121–141, 1982.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, pages 285–309, 1955.

[UR71] A. Urquhart and N. Rescher. *Temporal Logic*. Springer, 1971.

[Var01] Moshe Y. Vardi. Branching vs. Linear Time: Final Showdown. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–22, London, UK, 2001. Springer-Verlag.

[Var07a] Moshe Y. Vardi. Automata-theoretic model checking revisited, 2007.

[Var07b] Moshe Y. Vardi. From monadic logic to PSL, 2007. Paper in Festschrift for Boaz Trakhtenbrot.

[Var07c] Moshe Y. Vardi. The Büchi complementation saga. In *Proceedings of the 17th Symposiumon Theoretical Aspects of Computer Science (STACS07)*, Lecture Notes in Computer Science, pages 12–22, Aachen, Germany, February 2007. Springer.

[VW84] Moshe Y. Vardi and Pierre Wolper. Yet Another Process Logic (Preliminary Version). In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, pages 501–512, London, UK, 1984. Springer-Verlag.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 1994.