TOPICS IN

Structural Operational Semantics

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

ARNAR BIRGISSON

Supervisor: Luca Aceto

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

School of Computer Science
Reykjavík University
Kringlan 1, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6301
http://www.ru.is

Til afa,

fyrir að leyfa mér að fikta í tölvunni hans.

Contents

chapter one

Introduction

Languages are among the most important and ubiquitous concepts in Computer Science. In almost every sub-field of Computer Science, one can find specific languages for describing and reasoning about the concepts of that field. Programming languages are the best known example of course, but file formats, network protocols, instruction sets and even the various diagrammatic techniques used in Software Engineering can be considered as (visual) languages. Theoretical Computer Science has specification languages and various logics. Natural Language Processing has markup languages for describing voicing and sentence structure. Artificial Intelligence uses specific languages for describing behaviour, the rules of games and the constraints of planning problems. Indeed, it seems to be common practice in Computer Science to invent formalisms for specific problems, and these formalisms very often involve some kinds of languages.

Any language consists of two parts: its *syntax* and its *semantics*. The syntax defines what strings of symbols are valid, i.e. part of the language, while the semantics defines the actual *meaning* of any valid string. Formal specification of syntax is very common, even in non-academic use of Computer Science. However, one can argue that what really defines the true nature of a a language is its semantics. Different languages are much rather set apart by different semantics than different syntax. This thesis is a study, by way of example, of one specific technique of specifying semantics formally, namely *Structural Operational Semantics*. To put things in context, we'll start by an informal overview of this field.

1.1 Structural Operational Semantics

Structural Operational Semantics [Plotkin 2004a,b], SOS or simply Operational Semantics [1] for short, is a way of defining the meaning of terms in formal languages. By *formal languages* we mean any language for specifying ideas formally in the mathematical sense. This includes programming languages, process languages for modelling and verification as well as many others. By *terms* we usually mean programs or specifications written in these languages.

As the name indicates, SOS describes semantics in terms of program *structure* and the *operations* a program carries out as it computes. An SOS specification of the semantics for a certain language is a collection of rules. These rules specify how

[1]The term Operational Semantics is sometimes used as a synonym for Structural Operational Semantics (*small-step* semantics) and Natural Semantics (*large-step* semantics). In this thesis we are mostly concerned with the former.

a term with a certain structure behaves, by describing the operation that the next step of execution of this term performs (often on a hypothetical machine), and what is the term that should be executed for the next step after that. An example of an SOS rule is

$$\frac{t_1 \xrightarrow{a} t_1' \quad t_2 \xrightarrow{b} t_2'}{\mathsf{op}(t_1, t_2) \xrightarrow{b} t_2'} \tag{1.1}$$

This rule specifies how a term of the form $\mathsf{op}(t_1, t_2)$ behaves, where t_1 and t_2 can be any sub-terms as allowed by the syntax. This is seen by looking at the left-hand side of the *conclusion*, the part appearing below the line. The two expressions above the line are called *premises*. These are operations of the sub-terms that describe when the rule is applicable to derive a step of computation of the composite program $\mathsf{op}(t_1, t_2)$. This particular rule only applies if the operations in the premises can be deduced from the collection of rules. Naturally, this depends in each case on what the terms t_1 and t_2 actually are. If the operations in all the premises are valid for given sub-terms, the label of the arrow and the right-hand side below the line specify what is the operation performed by executing $\mathsf{op}(t_1, t_2)$ and what is the term that results after doing so. As one can see, both of these may be parameterised with information from the premises.

The hypothetical execution of terms proceeds by finding a rule that matches the term to be executed and whose premises are met. This rule then specifies an operation, i.e. *a single step* of execution, and the term to use for finding the next step. This process is repeated to create a sequence of operations. It is important to note that execution in this context does not necessarily mean execution on a real machine, but rather it is a useful abstract metaphor for reasoning about program behaviour. We say the sequences of operations are steps in the execution of a program on a hypothetical machine.

Often it is useful to indicate when such a sequence may stop, i.e. when the program terminates. We often do this by designating a specific set of terms as *terminal*, meaning that when a sequence of operations reaches such a term, the application of rules stops. Sometimes this is the empty term, e.g. a program of the form print "Hello"; print "World" might perform the following sequence of operations

$$\mathsf{print\ "Hello";\ print\ "World"} \xrightarrow{!"Hello"} \mathsf{print\ "World"} \xrightarrow{!"World"} \epsilon$$

where the operation $!string$ stands for writing $string$ to the screen, and ϵ is the empty program. In other settings the terminal terms may represent values. For example, a functional programming language might specify the meaning of the term $50 - 4 \times 2$ with the following sequence of operations.

$$50 - 4 \times 2 \longrightarrow 50 - 8 \longrightarrow 42$$

In this case, the term 42 is terminal since it contains no operators.

There is an important difference between the two approaches; in the former case the natural meaning of the program print "Hello"; print "World" is determined by the sequence of operations that its execution goes through, while in the latter the meaning of the program $50 - 4 \times 2$ is represented by the final value that the sequence reaches. Which one we choose depends on the particular setting in which we are using SOS.

For the latter interpretation, where the meaning of a term is taken to be the final value reached by a sequence of operations, there is an important thing to note about an SOS specification (collection of rules). In the general case, there is nothing that prevents the specification to contain rules that allow us to deduce *multiple* sequences of operations. For example, consider a system that contains the rule 1.1 above, but also contains the following rule.

$$\frac{t_1 \xrightarrow{a} t_1' \quad t_2 \xrightarrow{b} t_2'}{\mathsf{op}(t_1, t_2) \xrightarrow{a} t_1'} \tag{1.2}$$

Presented with a term of the form $\mathsf{op}(t_1, t_2)$, we can see that both rules may apply (depending on t_1 and t_2). If they do, we have a case of non-determinism[2] where the term may either be executed according to rule 1.1 or rule 1.2. In fact, an operator with this pair of rules is known as a *choice operator*, i.e. the execution of the term $\mathsf{op}(t_1, t_2)$ may choose whether it behaves like t_1 or like t_2.

It is not difficult to see that, when we take the meaning of a program to be its final value, if such non-determinism exists in the SOS specification, this meaning is not well defined. A term might give rise to multiple sequences of operations and thus multiple final values. Thus, when this view of meaning is taken, which is common when dealing with purely functional languages, we often make the requirement that the language's specification given by the rules is *deterministic*, i.e. for each term there is at most one operation and subsequent term that can be deduced from the collection of rules. Such collection of rules are the topic of Chapter 4 of this thesis.

In the other setting, where meaning of a term is taken to be the sequence of operations it gives rise to, non-determinism is generally allowed. This is for example the case in Process Algebra, where the meaning of a term is simply determined, in some formal sense, by the set of all possible behaviours it may generate. Two terms might for instance be considered equal if they generate the same set of operation sequences.

Sometimes the terms of the language don't contain enough information themselves to model execution. This is for example the case in programming languages that have variables which are globally bound. To find the value of a program term $3 + x$, one needs to know the value of x. In SOS specifications, this is solved by using *configurations* instead of terms in the rules. A configuration is a predefined structure which models the state of the execution completely. In the case of languages with variables, a common formulation is to represent the states as pairs of a term (with the same meaning as described above) and a *variable store*, written $\langle t, \Theta \rangle$. The variable store is in turn a function from the set of variables to actual values (or terms in the case of lazy languages). A typical rule in such a language

[2]Here we use the term *non-determinism* loosely. In process algebra, we only use this term if the labels in the conclusion of the two rules match. The key point here is that often there is a choice of several rules that can be applied to a particular term.

might look like this.

$$\frac{}{\langle \mathsf{x} := n, \Theta\rangle \longrightarrow \langle \epsilon, \Theta[x \mapsto n]\rangle} \qquad n \in \mathbb{N} \tag{1.3}$$

Note that this rule has no premises, which means that it applies whenever the term to be executed matches the left-hand side of the conclusion. The rule specifies that the operation of executing an assignment term, e.g. $\mathsf{x} := 28$, under a store Θ, results in a configuration with an empty term and a store that is identical to Θ except for its value in x, which is mapped to 28 (this is the conventional meaning of the $[\cdot \mapsto \cdot]$ syntax). Formally there is nothing special about using configurations instead of terms; configurations can themselves be considered "terms" of an extended language.

Another interesting thing to note about rule 1.3 is that it is in fact a *rule schema*. In other words, it represents a countably infinite number of rules, indexed by the natural number n. This is common when a part of the syntax of the language comes from a large domain such as $\mathbb{N}$.

This thesis consists of three main chapters, each of which is an independent paper. While their topics are in essence not related to each other, they all make use of operational semantics in a central manner. Although familiarity with SOS helps, the informal introduction above should provide the reader with enough background to read Chapter 2, which exemplifies a fairly complex use case of SOS. Chapters 3 and especially Chapter 4 use semantics in a more formal way; these chapters will each introduce the necessary preliminaries needed for their discussion. The following section introduces each chapter and highlights their ties to operational semantics.

1.2 Thesis contributions

Over the course of 12 months, rather than working solely on one single MSc study project, I have participated in several research projects at Reykjavik University and at the Technical University of Eindhoven. The result of this work are research contributions made by my co-authors and me to a few different fields of Computer Science. Each of these projects have built on the theory of SOS; in fact one of the projects (Chapter 4) is only about the theory of SOS rule systems, independent of their use.

The papers are arranged in order of increasing abstraction. Chapter 2 uses SOS to specify the semantics of an authorisation framework in a functional programming language. The SOS specification presented is non-trivial, but is intended solely for clarifying the semantics of this particular framework. The specification is accompanied by a detailed discussion of the semantics as well as an implementation of the framework in question.

Chapter 3 is in the field of Process Algebra. It uses SOS to provide quotienting techniques a la [Larsen and Xinxin 1991] for extensions to the process specification language CCS and Hennessy-Milner logic. CCS has a simple operational semantics and the paper proves, using the semantics, a powerful theorem for studying properties, that include past modalities, in a decompositional manner.

Finally, Chapter 4 goes one abstraction level above SOS specifications and provides so-called *meta-theorems* about rule systems that guarantee their determinism and the idempotence of certain operators. The meta-theorems consist of syntactic conditions on the rules themselves, such conditions are generally referred to as *SOS rule formats*.

In order to give the reader enough background for each chapter, we will now describe the general field of each paper, its contributions as well as highlight the specific contributions I made to each.

1.2.1 Semantics of Transactional Memory Introspection

Chapter 2 builds on previous work of [Birgisson et al. 2008]. In that paper we present an authorisation architecture called Transactional Memory Introspection, or TMI. The motivation for this architecture comes from the fact that Software Transactional Memory has recently become a popular way of avoiding race conditions in concurrent programs. Software Transactional Memory, or STM for short, tackles the issue of shared memory by replacing programmer managed locks with transactions. Where programmers would conventionally manage access to shared resources by careful lock placement, they may use STM instead to offload this responsibility to a machine controlled framework.

When using STM, programmers do away with lock management and instead mark sections of code as *atomic*. At run-time, an STM system will, as part of the program in question, ensure that the accesses a single thread makes to shared resources inside such atomic sections, appear atomic to other threads. Moreover, STM provides an isolation guarantee, that threads running inside such atomic sections do not affect, nor are affected by the actions of, concurrent threads. Semantically this is equivalent to enforcing a rule which says that only one thread may be running in an atomic section at each time, sometimes referred to as the *serializability* of transactions.

The beauty of STM comes from the fact that the actual implementation does not enforce such strict policies, as that would hurt performance. Instead, multiple threads are allowed to execute simultaneously inside atomic sections. Meanwhile, the STM system will carefully monitor the actions of each one of the threads. Generally, the threads will be accessing disjoint sets of resources, so most of the time this simultaneous execution poses no problems. However, in the cases where threads in atomic sections do conflict in their accesses, the STM system will notice and simply roll back some or all of the threads involved, and restart their execution at the start of their atomic sections. A rollback consists of undoing all work done by the threads, and will be triggered in cases when the execution of a thread has violated the isolation guarantee of the STM system. In practice, such violations happen in the minority of cases, so often the overhead of this approach will be paid for by the overhead saved in not using fine grained locking.

To implement the above, an STM system generally must provide

- isolation of concurrent threads in atomic sections,

- monitoring of resource access to detect conflicts,
- the ability to abort and rollback execution of an atomic section.

In [Birgisson et al. 2008] we argue that these mechanisms can be very beneficial to the problem of *policy enforcement*. Policy enforcement (or authorisation) is required in programs that handle sensitive data, to ensure that no illegal operations are performed, such as releasing confidential data or otherwise violate the applications policy. Traditionally this is done by careful code scrutiny and great effort on the programming side. Just as with locking, this practice is prone to errors.

Since programs that use STM systems for synchronisation purposes are already paying the price of monitoring and maintaining the ability to abort code, we conjectured that these mechanisms could be used to simplify policy enforcement at a relatively little extra cost. We identified three common problems (or errors) in modern policy enforcement code.

- *Time of check to time of use* (TOCTTOU) bugs. These happen when a policy decision is made prior to access, but the state used for the decisions is mutated in between by a concurrent thread.
- Difficulty in guaranteeing *complete mediation*, i.e. ensuring that any access, explicit or implicit, is accompanied by the relevant policy check. This is non-trivial in complex systems and empirical studies show that this is a source of several security holes in critical software.
- Difficulty in dealing with authorisation errors, when a policy violation has been detected, the system state must be carefully reset in order not to leak sensitive information or implicitly cause other policy violations.

The first of these is simply a synchronisation issue, and could be solved with locking. However, STM systems provide a synchronisation mechanism with added benefits; we can make use of its careful monitoring and abort capabilities to severely reduce the second and third difficulties.

When an application that makes use of TMI (which implies the use of STM) runs, any accesses made inside atomic sections are inspected by the STM system. TMI hooks into this inspection and also notifies an application specific security manager, which checks if the access is allowed by the application policy. At any time, the security manager has the capability to veto an access due to policy violation, in which case the abort mechanism of the STM is invoked. In one fell swoop this solves the issue of complete mediation, since the STM diligently inspects every access, as well as the issue of error handling since the rollback puts the system back into a consistent state and the isolation guarantees of the STM make sure that no concurrent thread gained knowledge of the actions leading up to the policy violation.

Our previous work of [Birgisson et al. 2008] consists of an extended discussion of the above, accompanied by a proof-of-concept implementation based on a prototype STM framework for Java [Herlihy et al. 2006]. However, while working with TMI and STM systems in general, we discovered that there are a great number of subtleties in the behaviour of unusual edge cases. An informal discussion, and even an implementation, did not provide a thorough understanding of the semantics of TMI. Thus the contribution presented in Chapter 2 consists of the formal specification of the semantics of our architecture, in the form of an extension to the semantics of the Haskell STM system [Harris et al. 2005]. The semantics is accompanied by a matching implementation.

My specific contributions to Chapter 2 consist of most of the technical work involved. I built the extension of the Haskell STM semantics, which went through several iterations of discussions with my co-author and revisions. In parallel I wrote the implementation in Haskell, which provided a lot of insight into the design decisions behind the semantic specification. I wrote the initial versions of most of the text, except for the introduction and the background on STM and TMI. All sections underwent a rewriting phase carried out jointly by my co-author and me.

This work has been accepted for publication in the proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS 2009), scheduled for June 15th 2009 in Dublin, Ireland.

1.2.2 Decompositional Reasoning about the History of Parallel Processes

In *model checking*, process specification languages are used to construct behavioural models of software and hardware systems, in particular reactive systems. These models are then used for detailed analysis of a system's behaviour. The process specification languages are usually accompanied by logic languages that allow the designer to specify an array of desirable, or non-desirable, properties. Such properties include

- *liveness properties,* guaranteeing that the system can always continue no matter the input it faces;
- *safety properties* such that the system will never perform certain critical operations unless it is in a state where it is safe to do so; and
- *security properties* such as ensuring that operations are properly authorised.

In most cases, the semantics of the specification languages is given as Structural Operational Semantics. A model is simply a term of the language, where the possible execution paths given by the semantics define the behavioural capabilities of the model. The properties to be checked are then described as logical expressions, often in logics which include modal operators (e.g. the model *can* perform a particular operation) or operators on the possible sequences of operations (such as *on each possible path there is a term with a particular property*). Examples of the former include Hennessy-Milner logic [Hennessy and Milner 1985] and of the latter Computation Tree Logic [Clarke and Emerson 1981] and Linear Temporal Logic [Pnueli 1977].

Most specification languages, or *process calculi*, include an operator that allows one to describe a process resulting from the parallel composition of two or more agents Usually this models interleaving concurrency, but in general a model of a process, which is composed of multiple parallel components, can exhibit the behaviour of any of the components in addition to certain synchronising behaviour where two or more of the components communicate. This means that the number of possible behaviours of a system grows exponentially in the number of parallel components.

Model checking is the act of testing if a certain model satisfies a property described by a logical expression. The resource usage of the algorithms that perform model checking depends directly on the number of possible behaviours that the model can exhibit, as the algorithms must exhaustively check every possible behaviour. Since many systems are best described as a number of smaller systems working in parallel such state space explosion is one of the biggest hurdles that must be overcome to make model checking practically useful.

One way of addressing the problem of the great increase in possible behaviours when systems are composed in parallel, is *decompositional reasoning*. Say we have a system made of the components P and Q composed in parallel, usually written as

$$\mathsf{System} = P \parallel Q.$$

We want to answer the following question: *Does System satisfies a given property described by a logic formula φ?* The number of transitions that the parallel term can afford can potentially be the product of the transitions afforded by each of P and Q, and even if P and Q alone are of moderate size, this product can be unmanageably large since a model checker must examine all the possibilities. However, in many cases we can use our knowledge of the system Q to construct another property (i.e. a logic formula) ψ such that the following question of P is equivalent to our original one of System: *Does P satisfy the property ψ?* In other words, if we can prove the bi-implication

$$P \parallel Q \text{ satisfies } \varphi \quad \Leftrightarrow \quad P \text{ satisfies } \psi,$$

we have reduced the size of the model checking problem to the size of P. However, the property ψ is constructed from φ *and* the system Q. The property ψ may thus be more complex than φ, and the size of Q has a direct impact on the difference. However this kind of reasoning has been shown to be efficient in model checking composed systems [Andersen 1995, Laroussinie and Larsen 1998].

Another use of decompositional reasoning arises when coupling it with synthesis of models from logical specifications. Suppose that a logical formula φ gives the specification of the expected behaviour of the system to be built and that our system has the form $P \parallel Q$. Assume furthermore that we have been given component Q off the shelf. A natural question to ask is whether we can we build P so that the resulting system will satisfy the specification φ. Using quotienting, this question can be reduced to whether we can build a model of the formula ψ. Such models can be constructed using known model construction techniques for many logics of interest.

Decompositional reasoning in essence tries to describe global properties of composed systems in terms of the local properties of its components, and dates back to the work of Larsen and Xinxin [Larsen and Xinxin 1991]. It has been further developed by several studies [Giannakopoulou et al. 2005, Xie and Dang 2006, Andersen 1995, Laroussinie and Larsen 1995]. Chapter 3 of this thesis aims to extend and apply such techniques in a setting which, to the best of my knowledge, has not been attempted before, namely the setting where the models maintain a record of their execution history.

In process calculus, one generally talks about *states* of execution. In the SOS sense, these states are the terms (or configurations) of the transition system generated by the semantic rules for the process calculus, starting from a designated initial state. The initial state is the term that represents the behavioural model of the system being described. Just as the initial state encodes the behavioural capabilities of the system, an intermediate state reached after performing some operations usually represents the behavioural capabilities of the system at that point. In particular, the intermediate states usually do not contain any information about the *past*, i.e. the operations performed by the system before reaching the aforementioned state.

For many properties that we want to check, this poses no problems. However, by enriching intermediary states with information about the past behaviour of the system in reaching them, some kinds of properties become easier to reason about. This includes for example epistemic properties, where one looks at the knowledge gained by agents (partially) observing the system during execution [Dechesne et al. 2007].

As for decompositional reasoning, the literature is rich with studies of process systems that involve the past [Hennessy and Stirling 1985, Phillips and Ulidowski 2006, Laroussinie and Schnoebelen 2000, Nicola et al. 1990]. Our contribution in Chapter 3 is combining the two fields. We build on a core subset of the process calculus CCS [Milner 1980] and extend the formalism based on its operational semantics to states that maintain full information about the execution history of processes. We similarly extend Hennessy–Milner Logic [Hennessy and Milner 1980] with modalities that look back in the history, in contrast with the standard HML modalities that look forward to the possible future behaviours. The main theorem of the chapter proves that decompositional reasoning can be applied to parallel processes in this setting.

The results presented here are however only a milestone towards a richer theory of decompositional reasoning about the past, albeit an important milestone. Work is currently underway to extend these results further, to include the ability to reason about recursive properties, i.e. those containing fixed-point operators, which greatly enhances the expressiveness of our extended logic.

The technical work and writing of Chapter 3 is for the most part mine apart from the introduction.

1.2.3 Rule Formats for Determinism and Idempotency

When designing a language, or analysing the semantics of existing ones, one is often interested in certain algebraic properties. For example, the language might contain an operator +, which combines two subterms in some way. Naming an operator + immediately signals to the user of the language that this operator possesses some properties of regular arithmetic addition, such as commutativity and associativity.

These are algebraic properties described by the equations

$$x + y = y + x \quad \text{and} \quad (x + y) + z = x + (y + z).$$

Such properties are often desirable to have, both to simplify the mental model one has of a language and also for technical purposes. For example, associativity can be very useful for automatically distributing large computations across an array of processors.

The properties we are are interested in are many. Besides commutativity and associativity, other useful properties include zero and unit elements of operators, and determinism and idempotency. In Chapter 4 we are concerned with the last two; determinism of operators, which means that at most one operation and subsequent term can be produced by the execution of a given term; and idempotency of operators, which is best described by the algebraic equation

$$f(x, x) = x$$

where f is an operator of the language. As noted in Section 1.1 earlier in this introduction, determinism is often a very important property. Idempotency is also a very natural requirement to make of certain operators.

Given an SOS specification of a language, such properties can be proven to hold. Such proofs generally consists of structural induction on the syntax of the language and/or on proof structures that arise when the SOS rules are used to deduce the operations that a term can afford. Often one has to consider a number of cases, and for real life languages, both the syntax and the number of rules can become reasonably large, so that often such proofs are tedious listings of a great number of cases. Such proofs are generally tedious to construct and check, as well as prone to errors. Furthermore such proofs must be made in the context of one language, and if the syntax or semantics of an evolving language change any existing proofs must be adapted and re-checked.

This has given rise to the *meta-theory of SOS*, in particular so-called *rule formats* [Aceto et al. 2001, Mousavi et al. 2007]. By putting constraints on the SOS rules used for a language (or a part of a language), one can prove properties such as those given above, in general for any language which as an SOS specification that meets the constraints. Often the constraints can be kept purely syntactical, in which case it becomes a relatively easy matter to check a certain specification against those constraints. Such constraints are called rule formats, and are accompanied by meta-theorems that state that any semantics that meets the constraints defines a language that has a particular property.

Many rule formats exist already. There are rule formats for commutativity [Mousavi et al. 2005] and associativity [Cranen et al. 2008] of operators, and congruence of behavioral equivalences [Verhoef 1995], as well as for less algebraic properties such as non-interference [Tini 2004] and stochasticity [Lanotte and Tini 2005]. In Chapter 4 we present two related formats. One guarantees the determinism of a transition system (or a subset thereof) and the other guarantees idempotency of a given operator.

My contributions to this work include both technical developments and writing. This work has been published in the 3rd International Conference on Fundamentals of Software Engineering (FSEN'09) in April 2009. The three proofs appearing in the chapter underwent peer-review through the FSEN program committee, but were omitted from the conference publication due to space constraints. Otherwise the published version is identical to the one that appears here.

1.3 Acknowledgements

The past year has been extremely interesting for me and has convinced me to pursue a career in Computer Science research. I feel privileged for the opportunities I have been granted to work on difficult and interesting problems, aside world-class researchers in their respective fields.

My sincerest thanks go to my supervisor, Luca Aceto, for sharing his great enthusiasm for computer science, and his extensive experience of research. He has always been available for me and willing to answer all of my questions, no matter how many times I have asked them before. Working with him has been a pleasure and an extremely valuable experience for which I am very grateful.

I would also like to thank Úlfar Erlingsson who, besides being a great teacher, has also become a good friend of mine through our long and enjoyable conversations. He has taken it upon himself to be my mentor in many ways, well beyond his duty, and from this I have benefited greatly. I owe Úlfar a great debt for actively singing my praises to many of the world-leading experts, giving me a head-start in the research community. It is now up to me to live up to them.

I thank MohammadReza Mousavi for generously hosting me for a visit to TU/e during the fall of 2008. Mohammad's drive is something I aspire for and his advice and confidence in me has helped me achieve things that I am proud of.

I also want to thank Anna Ingólfsdóttir and Michel Reniers for their support and enjoyable collaboration. I give special thanks to Maja Mei-Xin Aceto for commenting on the late drafts of this thesis with a drawing of a cheerful guy with six legs.

The work presented here has been partly supported by the projects "New Developments in Operational Semantics" (nr. 080039021) and "Topics in Structural Operational Semantics" (nr. 080890008) of The Icelandic Research Fund.

Thanks to Stefán Freyr, Hilmar and Pálmi for their friendship and for tolerating my constant interruptions. Thanks to Guðmundur Bjarni, for always being there with his support, friendship and honest opinion.

My greatest thanks go to my dearest friend, Hanna María. Without your encouragement and endless support, I would never have started this in the first place.

chapter two

Semantics of Transactional Memory Introspection

joint work with Úlfar Erlingsson

2.1 Introduction

The implementation of security enforcement mechanisms requires special care, as any flaw may open the door to malicious attacks. This is especially true in the case of multithreaded software, as the designer must consider all possible interleavings of code paths. In [Birgisson et al. 2008] we presented Transactional Memory Introspection (TMI), an architecture that greatly simplifies the implementation of correct reference monitors on mechanisms that implement Software Transactional Memory (STM) [Harris and Fraser 2003, Herlihy and Moss 1993] support. In this paper we present a formal semantics for TMI, as well as a reference TMI implementation over the Haskell STM. These specifications clarify the TMI architecture and help identify and resolve ambiguities in its implementation.

STM systems provide many useful guarantees that make the implementation of multithreaded software easier and less error-prone. In particular, STM offers atomicity and isolation through optimistically executing concurrent code and monitoring for conflicting accesses to resources. By providing rollback mechanisms, STM systems can resolve conflicting accesses by undoing the work of a transaction and retrying that transaction again, from the beginning.

All STM implementations must perform bookkeeping of accesses (such as reads and writes) to shared resources. By imposing on this bookkeeping, and the necessary monitoring and validation steps, TMI provides facilities to support the creation of robust and correct enforcement mechanisms. TMI provides *complete mediation* by enhancing the STM runtime checks against conflicting, concurrent accesses, and TMI adds the requirement that all accesses must have been successfully authorized before a transaction is committed. TMI also *simplifies error handling*. When unauthorized accesses are detected in a transaction, the transaction is rolled back and not retried. This saves the programmer from the onerous and error-prone task of performing clean-up after a failed authorization.

Another common problem in traditional authorization is *time of check to time of use* bugs. Such bugs arise when an authorization check is used to decide if a dangerous operation should be performed, and when the interleaving of code execution may cause state changes that invalidate that decision, before the dangerous operation is actually performed. TMI resolves this problem, by making use of STM mechanisms to execute both the authorization check and the dangerous operation within a single transaction.

In [Birgisson et al. 2008] we give a comprehensive, informal overview of the TMI architecture and also evaluate a Java TMI implementation built on the DSTM2 library [Herlihy et al. 2006]. In this previous, companion paper, we also discuss the relationship between TMI and other approaches, such as aspect-oriented and transactional techniques for security enforcement.

In this current paper, we give a more formal treatment of TMI, and provide a clear, well-defined structural operational semantics [Plotkin 2004a] for the TMI architecture. Our TMI semantics builds on the well founded semantics for the the Haskell STM system in [Harris et al. 2005], and is accompanied by an implementation over the Haskell STM system.

We found that the development of a formal semantics alongside an implementation helped us us disambiguate design choices and resolve ambiguities. In particular, the formal semantics allowed us to safely combine multiple TMI actions and different security managers into a single, atomic authorization decision. The implications of such compositionality are not clear, given only informal reasoning, and, indeed, some of our initial implementation strategies did not provide correct enforcement. However, as described further in Section 4, when combined with a formal semantics, we can establish that our TMI implementation correctly enforces the intended security policy.

The structure of the paper is as follows. In Section 2 we give the necessary background, including STM systems and how TMI makes use of their mechanisms, as well as an overview of the Haskell STM implementation. Section 3 covers TMI in greater detail and describes its Haskell implementation from the user standpoint. Section 4 defines the formal semantics of TMI, building on existing semantics for STM Haskell. Section 5 describes the key elements of our Haskell implementation and Section 6 discusses future work.

2.2 Background

2.2.1 STM and TMI

STM provides attractive guarantees for multithreaded software; namely atomicity, consistency and isolation of specifically marked blocks of code in *transactions*. In general, STM implementations must do so by performing

- careful monitoring of the resources that are accessed within a transaction,
- validation of the accesses of concurrent transactions, and
- complete rollback of the effects of aborted transactions.

TMI builds on this machinery and allows security enforcement to benefit from the STM guarantees. TMI helps the programmer to write correct enforcement mechanisms and simplifies error-handling. In [Birgisson et al. 2008] we outline three main benefits of TMI:

Complete mediation. TMI provides complete mediation by implicitly invoking the reference monitor before any effects of a transaction are permanently committed. The reference monitor validation checks are able to inspect the resource access logs of the STM and may veto the commit if an application specific policy is violated. In general, this requires that STM mechanisms provide strong atomicity, i.e. resources marked for transactional scrutiny may not be accessed outside the scope of a transaction.

Freedom from TOCTTOU bugs. *Time of check to time of use* (TOCTTOU) bugs arise in conventional enforcement mechanisms when interleaved threads may affect the policy decisions of each other. For example, a thread may make a policy-based decision to allow access to a certain resource, e.g. reading a memory location. Before that operation is actually performed, execution may be preempted by another thread. That thread can change the global state so that the policy decision becomes invalid, e.g. by writing privileged information into the memory location.

This problem is implicitly solved by using STM, which guarantees that transactions are isolated and cannot affect the policy decisions of each other.

Simplified error handling. In the event of an authorization failure, TMI uses the STM facilities to completely roll back the effects of the transaction in question and raise an appropriate exception to the code that initiated the transaction. This frees the programmer from having to undo state changes leading up to the unauthorized operation, a common source of errors [Weimer and Necula 2008].

2.2.2 Haskell STM

For a formal treatment, we build our semantics and implementation on those of the Haskell STM [Harris et al. 2005], which in turn is built on Concurrent Haskell [Peyton Jones et al. 1996]. Concurrent Haskell is an extension to Haskell 98, a lazy (i.e. call-by-name), pure, functional language. It supports concurrent threads and communications between them. Non-pure computations are modelled with *monads* [Peyton Jones and Wadler 1993]; this includes computations with side-effects such as input/output and mutable state.

The main entry to a Haskell program is an instantiation of the I/O monad, i.e. a value that represents an *action* of the type `IO ()`. An action of this type can, in addition to performing pure computation, perform other I/O actions by way of composing smaller actions into larger ones. For an example, Haskell standard libraries define the basic I/O actions `getChar` and `putChar`, which read from standard input and write to standard output, respectively. The most common composition is simple sequencing. For example the composed I/O action

```
main = do { c <- getChar; putChar c; putChar c }
```

defines an action that, when executed, will perform the three actions listed in sequence.

In general a value of type `IO a` represents an action that when executed, may perform some I/O operations as defined by the Haskell libraries and then result in a value of type `a`. Pure functions cannot execute such actions without jeopardizing their purity and this is neatly enforced by the Haskell type system. Naturally, I/O actions are however free to run pure computations. Thus the only way to get at the value of an action is if it is a part of a bigger I/O action. The Haskell runtime bootstraps the whole process by executing the special I/O action called `main`.

In addition to conventional input and output, I/O actions can perform reads and updates of mutable memory cells. The type `IORef a` represents a mutable cell that contains a value of type `a`. Haskell provides the basic I/O actions `newIORef`, `readIORef` and `writeIORef` for manipulation of such cells. As with other I/O actions, these operations can only be used when composing larger I/O actions.

Concurrent Haskell supports explicit forking of threads through the I/O action `forkIO`.

```
forkIO :: IO a -> IO ThreadID
```

`forkIO` takes another I/O action as a parameter and spawns a new thread to execute the action, immediately returning a newly allocated thread identifier. For further discussion of concurrency we refer to [Peyton Jones 2001] or tutorials such as [Peyton Jones and Singh 2008].

The Haskell STM is based on a monadic type similar to the one for I/O actions, namely `STM a`. A value of this type represents an *STM action*, which when executed may perform smaller STM actions and result in a value of type `a`. STM actions may contain pure computations as well, but note that they *cannot* contain e.g. I/O actions. The main STM actions provided are actions that allow manipulations of another kind of memory cells, which have the type `TVar a`, where `a` is the type of value that the cell holds. The actions are `newTVar`, `readTVar` and `writeTVar`, so they have the same power as their I/O counterparts. The important thing to note is that sets of `IORefs` and `TVars` are kept separate; one can only be used in I/O actions and the other in STM actions.

STM actions can be composed. Similar to I/O actions, the most common composition is sequencing, but in addition Haskell STM provides the basic STM action `retry` and a combinator `orElse`. The action `retry` is a blocking operation for STM actions, which restarts the current transaction with potentially updated `TVar` contents. By issuing `retry`, the programmer is stating that the current transaction cannot finish for the state of `TVars` it started in. The Haskell STM provides an optimized implementation of `retry`. This implementation captures the set of `TVars` that a transaction has read before the retry, and suspends the transaction until at least one of those `TVars` has been updated. This makes sense because the *only* outside factors that can affect the execution of an STM action are the values of the `TVars` it reads.

If `t1` and `t2` are STM actions, then ``t1 `orElse` t2`` is an STM action that first tries performing `t1` on its own. If `t1` invokes the `retry` action, then the combined action rolls back the effects of `t1` and tries `t2` instead. If that one retries also, the whole action retries, but waits for updates on the variables read by *both* `t1` and `t2`.

For an example how the above can be used for synchronization primitives such as communication channels, see Section 4 of [Harris et al. 2005].

While the basic STM actions and their compositions give us a way to build larger STM actions, we have not discussed how those actions can be run or how they relate to transactions. For this, STM Haskell provides us with the `atomically` function,[1] whose type is

```
atomically :: STM a -> IO a
```

This function gives an I/O action that, when performed, will execute the input STM action. The atomicity comes from the fact that STM Haskell will guarantee that the effects that the STM action has on `TVars` are atomic, i.e. they all become visible at once and that what happens inside the STM action is not affected by concurrent threads.

The Haskell STM system does this by monitoring concurrent invocations of STM actions, taking care of rolling them back if they conflict with each other and retrying them. As an example, the following program creates a transactional variable holding a counter and spawns three threads that each increments the counter atomically.

```
increment :: TVar Int -> STM ()
increment counter = do x <- readTVar counter
                       writeTVar counter (x + 1)

main = do c <- atomically (newTVar 0)
          forkIO (atomically (increment c))
          forkIO (atomically (increment c))
          forkIO (atomically (increment c))
```

The `increment` action is a classical example of where a race condition might occur in a traditional setting, but in our situation the STM system will guarantee the atomicity of each invocation.

2.3 Transactional Memory Introspection

In this section we give an overview of the TMI architecture and how it is implemented. We then describe our Haskell implementation from a user standpoint.

2.3.1 Overview of TMI

As described in our previous work [Birgisson et al. 2008], the TMI architecture aims to raise the level of abstraction in the implementation of security enforcement mechanisms. It allows the programmer to decouple application logic from security enforcement. Just as STM frees the programmer from worrying about lock acquisition order and other synchronization efforts, TMI can be used to eliminate concerns about check placement, race conditions and exceptional execution paths.

TMI provides these guarantees by imposing on the STM system. The programmer marks certain variables as security sensitive. This implicitly indicates to the

[1] While [Harris et al. 2005] uses the name `atomic`, the actual implementation of the Glasgow Haskell Compiler uses `atomically`.

STM system that these variables are shared, and ensures that the STM system will protect against race conditions in accesses to the variables. TMI enhances the monitoring of these security sensitive variables by ensuring that an access-control reference monitor is invoked every time that the variables are accessed.

Time of policy evaluation with TMI: The TMI architecture only loosely constrains when a policy must be evaluated, and in [Birgisson et al. 2008] we consider a number of alternatives. In particular, TMI enforcement can be *eager* or *lazy*. With *eager* enforcement, every access to a variable triggers the reference monitor, which immediately checks it against the relevant policy. If authorization is denied, the transaction is immediately aborted. With lazy enforcement, accesses to variables are simply logged (often they are already logged by the STM) and the logs are inspected by the reference monitor only at the end of the transaction. If any of the logged accesses are invalid, the whole transaction is aborted.

A key property of TMI enforcement is that policy decisions can be evaluated at any time, as long as they are evaluated in a serialized fashion, and evaluation is fully complete before the transaction commits. A good STM system will ensure that each transaction is executed in isolation, such that aborting one will have the same semantics as not having started it. This said, for our formal treatment and Haskell implementation, we focus on lazy enforcement only. Thus, the following discussion only deals with the lazy variant unless otherwise noted.

Utilizing TMI enforcement: To use TMI, the programmer declares a set of variables as security relevant. This implicitly indicates to the underlying STM system that those variables should be protected against race conditions. This means the values held by these variables can only be read or modified within a transaction, and that the STM system takes care of resolving conflicting accesses by concurrent transactions. This also means that, upon every variable access, TMI appends information identifying the variable in question to a transaction-specific *introspection log*. In particular, the introspection log will contain information about the creation, reading, and writing of the security sensitive variables.

In addition, TMI requires that all sections of code that access security-sensitive variables must be explicitly marked as *atomic*. To execute such atomic code sections, programmers initiate a TMI transaction and provide a reference to the atomic block and a *security manager*. The security manager is a block of code (or closure) that encodes the intended, application-specific security policy, and is able to determine whether a transaction introspection log satisfies the security policy. The security manager closure includes the active principal, and other auxiliary information that is needed to check policy compliance.

Finally, transaction commit plays a special role in TMI enforcement. TMI runs atomic blocks as transactions in the underlying STM system, but changes the semantics of transaction commit. After a TMI atomic block has finished execution, but before it is committed, TMI ensures that the security manager has fully evaluated whether the transaction introspection log complies with the intended security policy. Importantly, this evaluation occurs within the same STM transaction as the execution of the atomic block, and a commit of the transaction is attempted only if the security manager returns success.

Even when the transaction has complied with the security policies, the attempted commit may still fail, and the transaction is retried, in the case when the STM system finds conflicting concurrent accesses. Also, if the security manager finds the transaction in violation of policy, all state changes are rolled back—including changes to the security manager state, in the case of history-based policies—and, instead of retrying the transaction, an exception is raised to the invoker of the atomic block.

A simple example: The following pseudo code shows what software that makes use of TMI-based security enforcement might look like. (Note that the code makes use of function-argument currying.)

```
declare sensitive accounts = array of Account

function withdraw(account, amount):
    account.balance = account.balance - amount

function security_manager(user, log):
    if log contains <withdrawal from account>:
        if account.owner == user:
            return Allowed
    return Denied

main program:
    user = aquire_login_credentials()
    try:
        transaction with security_manager(user):
            withdraw(get_account(123456), 42)
    catch AuthorizationFailed:
        tell user about error
```

In this code, two aspects are especially noteworthy. First, security enforcement code is completely decoupled from the application logic and the function `withdraw` performs no authorization. Even so, complete mediation is ensured, since the introspection log is implicitly updated by the TMI reference monitor upon each access to account variables.

Second, in the case of authorization failure (e.g. a withdrawal from a different user's account), the error handler need only consider how to indicate the error to the user. The error handler need not clean up any mess: the state changes that happened during the transaction (if any) have already been rolled back when the error handler starts execution. Although perhaps not apparent in this simplified example, there is ample evidence that writing correct cleanup code is difficult, especially when multiple security-relevant operations are involved [Weimer and Necula 2008].

$$\begin{array}{rcl}
x, y & \in & \mathit{Variable} \\
r, t & \in & \mathit{Name} \\
c & \in & \mathit{Char} \\
 & & \\
V & ::= & r \mid c \mid \backslash x\texttt{->}M \\
 & \mid & \texttt{return}\ M \mid M \texttt{>>=} N \\
 & \mid & \texttt{putChar}\ c \mid \texttt{getChar} \\
 & \mid & \texttt{throw}\ M \mid \texttt{catch}\ M\ N \\
 & \mid & \texttt{retry} \mid M\ \texttt{`orElse`}\ N \\
 & \mid & \texttt{forkIO}\ M \mid \texttt{atomically}\ M \\
 & \mid & \texttt{newTVar}\ M \\
 & \mid & \texttt{readTVar}\ r \mid \texttt{writeTVar}\ r\ M \\
 & \mid & \texttt{newTMIVar}\ N\ M \\
 & \mid & \texttt{readTMIVar}\ r \mid \texttt{writeTMIVar}\ r\ M \\
 & \mid & \texttt{authorized}\ N\ M \mid \texttt{liftSTM}\ M \\
 & \mid & \texttt{getlog} \mid \texttt{UnauthorizedError} \\
 & & \\
M, N & ::= & x \mid V \mid M\ N \mid \ldots
\end{array}$$

Figure 2.1: Syntax of values (V) and terms (N, M)

While the above observations form the two main benefits of the TMI architecture, the third is freedom from TOCTTOU bugs. Without TMI, this example code might suffer from TOCTTOU race conditions, e.g., if accounts could change owners. However, with TMI, such account-ownership changes would be isolated, and a transaction would be guaranteed to see the same owner throughout its execution.

2.3.2 TMI in Haskell

We saw earlier how Concurrent Haskell uses the type system to confine operations on shared variables to STM actions, and provides a single function to wrap STM actions into an atomic I/O action. For TMI, we do something very similar. We confine operations on security sensitive variables to *TMI actions*, and provide a single function to turn a TMI action into an STM action and associating it with a security manager at the same time.

Figure 2.1 shows the extensions of STM Haskell with the TMI extensions (highlighted). We define a new monad that represents TMI actions and operations on sensitive variables. In addition, we lift all standard STM functions to their TMI counterparts. This is done so that an existing Haskell STM program can be easily adapted to TMI with minimal changes to their code. The TMI monad also encapsulates state, namely the introspection log of a transaction. The introspection log contains entries which specify the access type (create, read or write) of a variable and the *security descriptor* of a variable. Security descriptors are provided by the programmer when she creates sensitive variables and contain the metadata about the variable that is necessary for authorization, such as the owner of an account, permissions of a file, etc.

Since the type of security descriptors is application specific, our new monad type is polymorphic,

```
TMI d a
```

where `d` is the type of descriptors and `a` is the type returned by the action. For security sensitive variables, we have a type similar to `IORef a` and `TVar a`,

```
TMIVar d a
```

An instance of this type is a cell with a security descriptor of type `d` and a value of type `a`. While the value can change over time, the security descriptor is specified when the cell is created and cannot change after that. Creation, reading and writing of cells is performed with the following set of functions.

```
  newTMIVar :: d -> a -> TMI d (TMIVar d a)
 readTMIVar :: TMIVar d a -> TMI d a
writeTMIVar :: TMIVar d a -> a -> TMI d ()
```

For an example, the following code defines a descriptor type for a bank account. The account itself is represented by a simple integer.

```
-- Security descriptor for accounts
data AccountDescr = AccountDescr {
    acctOwner  :: String,
    acctNumber :: Int
}
type Account = TMIVar AccountDescr Int

createAccount :: String -> Int -> Int
                              -> TMI Account
createAccount owner number balance =
    newTMIVar (AccountDescr owner number) balance
```

The next function demonstrates reading and writing of the security-relevant account variables.

```
deposit :: Account -> Int -> TMI AccountDescr ()
deposit acct amount =
    do balance <- readTMIVar acct
       writeTMIVar acct (balance + amount)
```

To turn a TMI action into an STM action, we need to associate it with a security manager, i.e. a boolean function that evaluates the transaction introspection log of security-relevant accesses and determines if the transaction should be aborted. As an input to this function, TMI defines the type of an introspection log.

```
data AccessType = CreateVar | ReadVar | WriteVar
type TMILog d = [(AccessType, d)]
```

To specify the application specific policy, the programmer must supply the security manager, a function of the type `TMILog d -> Bool`. This function, along with a

TMI action is passed to the `authorized` function. The simplest security manager is one that performs no authorization and simply allows all operations.

```
allowAll :: forall d. TMI d a -> STM a
allowAll tx = authorized (const True) tx
```

A slightly more complex example is a security manager that looks at all `Accounts` touched by a transaction and verifies that they belong to the current user. The current user is passed to the security manager as the first argument, and this currying ensures that we satisfy the type required by `authorized`.

```
auth :: String -> TMILog AccountDescr -> Bool
auth user thelog = all checkowner thelog
    where
      checkowner :: (AccessType, AccountDescr)
                 -> Bool
      checkowner (_,descr) =
            user == (acctOwner descr)

-- Defined by the TMI module:
-- authorized :: (TMILog d -> Bool)
--            -> TMI d a
--            -> STM a

main =
  do acct <- atomically (allowAll mkAccount)
     atomically (doDeposit acct "alice")  -- OK
     atomically (doDeposit acct "bob")    -- FAILS
  where
     mkAccount = createAccount "alice" 123456 0
     doDeposit acct user =
         authorized (auth user) (deposit acct 42)
```

Since TMI actions are ultimately executed as STM actions, we also provide a lifting operation to lift STM operations into TMI operations, `liftSTM`. This allows for the embedding of an STM action inside a TMI action. Once the TMI action is turned into an STM action via `authorized`, the embedded action is just composed with it in the normal way. An interesting effect of this is that it allows for nested calls to `authorized`. While this might cause ambiguity for other implementations, in this Haskell-based implementation such nesting has clear and well-defined semantics, and can therefore be permitted. In fact, we will make explicit use of such nesting in the following sections to implement *privilege amplification*.

TMI actions are also composable in the same way STM actions are. This means the monadic bind acts as sequential composition and we provide `orElseTMI` and `retryTMI` that behave as their STM counterparts. When TMI actions composed with `orElseTMI` are turned into STM actions, via `atomically`, the security manager only sees the log entries for `TMIVar`-actions that are actually committed or could have affected the committed actions.

Thread soup	P, Q	$::=$	$M_t \mid (P \mid Q)$
Descriptors	$D_\perp$	$::=$	$M \cup \{\perp\}$
Heap	Θ	$::=$	$r \hookrightarrow M \times D_\perp$
Allocations	Δ	$::=$	$r \hookrightarrow M \times D_\perp$
Access types	T	$::=$	$\{\textsc{create}, \textsc{read}, \textsc{write}\}$
Log	Σ	$::=$	list monoid $([], \oplus)$ over $T \times D$
Evaluation	$\mathbb{E}$	$::=$	$[\cdot] \mid \mathbb{E} \mathbin{>\!>\!=} M \mid \texttt{catch}\ \mathbb{E}\ M$
contexts	$\mathbb{S}$	$::=$	$[\cdot] \mid \mathbb{S} \mathbin{>\!>\!=} M$
	$\mathbb{P}$	$::=$	$\mathbb{S}_t \mid (\mathbb{P} \vert P) \mid (P \vert \mathbb{P})$
Action	a	$::=$	$!c \mid ?c \mid \epsilon$

Figure 2.2: Program state and evaluation contexts

2.4 Formal semantics of TMI

To formalize the semantics of TMI, we build on the semantics for the Haskell STM presented in [Harris et al. 2005]. The semantics is a structural operational semantics in the style of Plotkin [Plotkin 2004a]. For the sake of completeness and to help the reader understand our extensions, we give a cursory explanation of the concepts of the semantics from [Harris et al. 2005] so that a reader not familiar with it may understand our extensions.

It is not obvious that the ideas presented in the previous section are indeed always safe. For example, we could not be sure that the nesting of TMI actions inside an STM action — in turn lifted to yet another TMI action — would result in a reasonable behavior. The construction of the following semantics greatly clarified our understanding of such subtleties. Our first drafts of the semantics revealed several ambiguities that were later resolved. Furthermore, constructing semantics for our intermediate implementation ideas sometimes revealed cases where incorrect behavior was possible, and the intended security guarantees of TMI were violated. Initially, for instance, a clear distinction of security-relevant data was missing and, while TMI actions naturally supported STM functionality, we had to explore several options to find the support for flexible STM and TMI combinations that is present in the final semantics.

In particular, the final semantics provides a a clear separation between TMI and STM actions, which allows STM actions to be lifted to the TMI level and ensures correct behavior when nesting TMI and STM actions, or several TMI actions, one within another. This nesting support provides powerful composability properties, and makes it possible to safely combine multiple TMI actions and different security managers into a single, atomic authorization decision. Without a formal semantics, the implications of such composability would have been unclear, and its correctness suspect. However, in our final semantics, given below, it is straightforward to see that the TMI security policy enforcement guarantees are always correctly maintained.

Figures 2.3 through 2.5 give the operational rules that describe the steps a program may take. At the top level, a program transforms a state of the form $P; \Theta$

via a labelled transition.

$$P;\Theta \xrightarrow{a} Q;\Theta'$$

P represents a program term in the syntax of Figure 2.1 while Θ stands for a memory store, a partial function from variable names to annotated terms. An annotated value is a tuple (t, d) where t is a program term and d is a value that holds the security relevant description of the relevant variable. The labels on transitions represent the program's input and output actions. Q and Θ' represent the term that is left unevaluated and the updated store after a transition, respectively.

To model atomicity of transactions, separate relations represent the top level I/O transitions and the STM actions. We extend this by adding a third relation representing the security relevant TMI actions. Furthermore we add a simple relation for evaluation of security managers under the context of an immutable transaction log.

Execution of a program proceeds by non-deterministically picking a program term from a collection of terms, each representing a separate thread of execution. One I/O transition of this term combined with the current store is performed and then the process is repeated. This models interleaved concurrency at the level of I/O transitions. STM transitions however can only be performed as a required premise of the `atomically` operator at the I/O level, and thus appear in this model as a single atomic step.

As mentioned in [Harris et al. 2005] there is no need to represent rollback, but contrary to the semantics in that paper, our extensions do need to formalize the notion of the transaction log as it is no longer purely an implementation detail. For simplicity though, we only model the log for security sensitive operations as they are the only ones relevant to the semantics of TMI.

2.4.1 Syntax, states and evaluation contexts

The syntax of terms for a subset of STM Haskell is given in Figure 2.1 with our TMI-related extensions (highlighted). Terms and values are standard except that the application of some monadic operators are considered values, a technique again lifted from [Harris et al. 2005]. The **do**-notation used up until now is standard syntactic sugar for the monad bind and return operations.

$$\begin{array}{rcl}
\textbf{do}\ \{x \texttt{<-} e\texttt{;}\ Q\} & \equiv & e \texttt{ >>= } (\texttt{\\}x \texttt{ -> } \textbf{do}\ \{Q\}) \\
\textbf{do}\ \{e\texttt{;}\ Q\} & \equiv & e \texttt{ >>= } (\texttt{\\\_} \texttt{ -> } \textbf{do}\ \{Q\}) \\
\textbf{do}\ \{e\} & \equiv & e
\end{array}$$

Figure 2.2 defines some symbols used in the semantics. The metavariable D represents a set of terms used to describe the security properties of variables. We extend this set with an invalid value $\perp$ and write $D_\perp$ for the extended set. A state of a computation is a pair (M, Θ) of a term that remains to be evaluated and a store Θ. The store maps variable names to terms and their variable descriptors. If a variable does not have a suitable descriptor, we use $\perp$ as a fill-in. This is used to distinguish security-relevant variables from other variables.

The set of access types, T, consists of three constants, each representing an operation performed on variables. An introspection log Σ is a list monoid of pairs

$$\boxed{\text{Administrative transitions} \qquad M \to N}$$

$$\begin{array}{rcll}
M & \to & V \quad \text{if } \mathcal{V}[\![M]\!] = V \text{ and } M \not\equiv V & \textit{(EVAL)} \\
\texttt{return } N \mathbin{\texttt{>>=}} M & \to & M\,N & \textit{(BIND)} \\
\texttt{throw } N \mathbin{\texttt{>>=}} M & \to & \texttt{throw } N & \textit{(THROW)} \\
\texttt{retry} \mathbin{\texttt{>>=}} M & \to & \texttt{retry} & \textit{(RETRY)}
\end{array}$$

$$\boxed{\text{I/O transitions} \qquad P;\, \Theta \xrightarrow{a} Q;\, \Theta'}$$

$$\begin{array}{rclll}
\mathbb{P}[\texttt{putChar } c];\, \Theta & \xrightarrow{!c} & \mathbb{P}[\texttt{return ()}];\, \Theta & & \textit{(PUTC)} \\
\mathbb{P}[\texttt{getChar }];\, \Theta & \xrightarrow{?c} & \mathbb{P}[\texttt{return } c];\, \Theta & & \textit{(GETC)} \\
\mathbb{P}[\texttt{forkIO } M];\, \Theta & \to & (\mathbb{P}[\texttt{return } t] \mid M_t);\, \Theta & t \notin \mathbb{P}, \Theta, M & \textit{(FORK)} \\
\mathbb{P}[\texttt{catch (return } M\texttt{) } N];\, \Theta & \to & \mathbb{P}[\texttt{return } M];\, \Theta & & \textit{(CATCH1)} \\
\mathbb{P}[\texttt{catch (throw } M\texttt{) } N];\, \Theta & \to & \mathbb{P}[N\,M];\, \Theta & & \textit{(CATCH2)}
\end{array}$$

$$\frac{M \to N}{\mathbb{P}[M];\, \Theta \to \mathbb{P}[N];\, \Theta} \ \textit{(ADMIN)}$$

$$\frac{M;\, \Theta, \{\} \overset{*}{\Rightarrow} \texttt{return } N;\, \Theta', \Delta'}{\mathbb{P}[\texttt{atomically } M];\, \Theta \to \mathbb{P}[\texttt{return } N];\, \Theta'} \ \textit{(ARET)}$$

$$\frac{M;\, \Theta, \{\} \overset{*}{\Rightarrow} \texttt{throw } N;\, \Theta', \Delta'}{\mathbb{P}[\texttt{atomically } M];\, \Theta \to \mathbb{P}[\texttt{throw } N];\, \Theta \cup \Delta'} \ \textit{(ATHROW)}$$

Figure 2.3: Evaluation of terms and monad operations and IO actions

(t, d) where t is an access type and d is a descriptor term; we use $[\,]$ for the empty list and $\oplus$ for concatenation, and in the semantics we use $[\cdot]$ as a constructor. Other symbols are conventional and taken from [Harris et al. 2005].

For a (partial) function f whose co-domain is a cross-product of two or more sets, and an integer i, we write f_i instead of $\pi_i \circ f$ where π_i is the standard i-th projection function. For convenience, we introduce the following notation for filtering logs. If Δ is a store and Σ is an introspection log, we define the Δ-*restriction of* Σ, indicated by $\Sigma|_\Delta$, thus

$$\begin{array}{rl}
[\,]|_\Delta & = [\,] \\
([(t,d)] \oplus \Sigma')|_\Delta & = \begin{cases} [(t,d)] \oplus \Sigma'|_\Delta & \text{if } d \in \operatorname{img}(\Delta_2) \\ \Sigma'|_\Delta & \text{otherwise} \end{cases}
\end{array}$$

Intuitively, $\Sigma|_\Delta$ is the list of entries from Σ which apply to variables defined by Δ, where variables are identified by their security descriptors.

Interleaving of operations is modelled with the evaluation context $\mathbb{P}$, often referred to as a *thread soup*. Through this evaluation context the semantics can non-deterministically choose a term for reduction from the parallel construct, each term representing a thread. Haskell terms are usually reduced according to the evaluation context $\mathbb{E}$, which allows for reductions of the right hand side of the >>= operator as well as within the body of a **`catch`** term. However, since we want to handle exceptions in a specific manner for STM and TMI actions, we will use the simpler context $\mathbb{S}$ which requires the operational semantics rules to specify explicitly how **`catch`** terms are handled.

2.4.2 Operational semantics

Figures 2.3 through 2.5 detail the transition relations of our semantics. Figures 2.3 and 2.4 are mostly the same as in the semantics of [Harris et al. 2005], parts added for TMI are indicated with a darker ink. The semantics uses several different transition systems that are layered such that a sequence of reductions in one layer becomes one reduction in the next layer above. This makes a sequence of transitions in a lower layer appear as one atomic transition at the higher level. There are three main layers - the top level I/O context, the STM context and the TMI context. An auxiliary transition system is used to reduce authorization functions.

Values and I/O transitions: The *admin* transitions of Figure 2.3 define the evaluation of terms to values via a function $\mathcal{V}$. This function is standard and its definition omitted here. Administrative transitions also include the behaviour of the monadic bind operator >>=.

The top level I/O actions are described by the labelled $\rightarrow$ relation. They operate on the $\mathbb{P}$ context, which allows for picking any program term from the thread soup for reduction. The first two rules are I/O primitives. The rule *FORK* is used to create a new thread and enter it into the thread soup, choosing a fresh thread id t. The rules *CATCH1* and *CATCH2* deal with exception handling as described in the appendix of the post-publication, extended version of the Haskell semantics [Harris et al. 2005]. The *ADMIN* rule allows for lifting of administrative transitions to the I/O transition relation. This is done to reduce repetition, as the administrative rules also apply to the STM and TMI transition relations, which have a similar lifting rule. Finally, the rules *ARET* and *ATHROW* enable the use of the `atomic` combinator to lift a sequence of reductions in the STM transition relation to a single I/O transition.

If the series of STM reductions results in a **`return`** value, the effects on the store are retained. If it however results in an exception (i.e. a `throw` value), the modifications to existing variables are discarded but any new allocations are retained. This is necessary as the exception value may hold references to newly allocated variables.

STM transitions: The STM transitions define the behaviour of STM actions. The states used in these transitions are extended from the I/O transitions by adding a separate store for new allocations Δ, and an introspection log Σ. A transition of the form

$$M;\, \Theta, \Delta, \Sigma \;\Rightarrow\; N;\, \Theta', \Delta', \Sigma'$$

represents a reduction of the term M to the term N. Some variables in Θ may be introduced or altered to yield Θ'. Δ is a store similar to Θ, that only tracks newly allocated variables while Σ is a log of accesses to TMI variables. Δ and Σ

$$\boxed{\text{STM transitions} \qquad M;\, \Theta, \Delta, \Sigma \;\Rightarrow\; N;\, \Theta', \Delta', \Sigma'}$$

$$\begin{array}{rcll}
\mathbb{S}[\texttt{readTVar}\ r];\, \Theta, \Delta, \Sigma & \Rightarrow & \mathbb{S}[\texttt{return}\ \Theta_1(r)];\, \Theta, \Delta, \Sigma & \\
 & & \quad \text{if } r \in \text{dom}(\Theta) \text{ and } \Theta_2(s) = \bot & (\textit{READ}) \\
\mathbb{S}[\texttt{writeTVar}\ r\ M];\, \Theta, \Delta, \Sigma & \Rightarrow & \mathbb{S}[\texttt{return}\ ()];\, \Theta[r \mapsto (M, \bot)], \Delta, \Sigma & \\
 & & \quad \text{if } r \in \text{dom}(\Theta) \text{ and } \Theta_2(s) = \bot & (\textit{WRITE}) \\
\mathbb{S}[\texttt{newTVar}\ M];\, \Theta, \Delta, \Sigma & \Rightarrow & \mathbb{S}[\texttt{return}\ r];\, \Theta[r \mapsto (M, \bot)], \Delta[r \mapsto (M, \bot)], \Sigma & \\
 & & \quad r \notin \text{dom}(\Theta) & (\textit{NEW})
\end{array}$$

$$\frac{M \to N}{\mathbb{S}[M];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[N];\, \Theta, \Delta, \Sigma} \quad (\textit{AADMIN})$$

$$\frac{M_1;\, \Theta, \Delta, \Sigma \;\overset{*}{\Rightarrow}\; \texttt{return}\ N;\, \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1\ \texttt{'orElse'}\ M_2];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[\texttt{return}\ N];\, \Theta', \Delta', \Sigma'} \quad (\textit{OR1})$$

$$\frac{M_1;\, \Theta, \Delta, \Sigma \;\overset{*}{\Rightarrow}\; \texttt{throw}\ N;\, \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1\ \texttt{'orElse'}\ M_2];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[\texttt{throw}\ N];\, \Theta', \Delta', \Sigma'} \quad (\textit{OR2})$$

$$\frac{M_1;\, \Theta, \Delta, \Sigma \;\overset{*}{\Rightarrow}\; \texttt{retry};\, \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1\ \texttt{'orElse'}\ M_2];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[M_2];\, \Theta, \Delta, \Sigma} \quad (\textit{OR3})$$

$$\frac{M;\, \Theta, \{\}, [] \;\overset{*}{\Rightarrow}\; \texttt{return}\ M';\, \Theta', \Delta', \Sigma'}{\mathbb{S}[\texttt{catch}\ M\ N];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[\texttt{return}\ M'];\, \Theta', \Delta \cup \Delta', \Sigma \oplus \Sigma'} \quad (\textit{XSTM1})$$

$$\frac{M;\, \Theta, \{\}, [] \;\overset{*}{\Rightarrow}\; \texttt{throw}\ M';\, \Theta', \Delta', \Sigma'}{\mathbb{S}[\texttt{catch}\ M\ N];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[N\ M'];\, \Theta \cup \Delta', \Delta \cup \Delta', \Sigma \oplus (\Sigma'|_{\Delta'})} \quad (\textit{XSTM2})$$

$$\frac{M;\, \Theta, \{\}, [] \;\overset{*}{\Rightarrow}\; \texttt{retry};\, \Theta', \Delta', \Sigma'}{\mathbb{S}[\texttt{catch}\ M\ N];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[\texttt{retry}];\, \Theta, \Delta, \Sigma} \quad (\textit{XSTM3})$$

$$\frac{M;\, \Theta, \{\}, [] \;\overset{*}{\rightharpoonup}\; \texttt{return}\ M';\, \Theta', \Delta', \Sigma' \qquad \Sigma' \vdash N \overset{*}{\rightsquigarrow} \texttt{return}\ N'}{\mathbb{S}[\texttt{authorized}\ N\ M];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[\texttt{return}\ M'];\Theta', \Delta', \Sigma} \quad (\textit{AURET1})$$

$$\frac{M;\Theta, \{\}, [] \;\overset{*}{\rightharpoonup}\; \texttt{throw}\ M';\, \Theta', \Delta', \Sigma' \qquad \Sigma' \vdash N \overset{*}{\rightsquigarrow} \texttt{return}\ N'}{\mathbb{S}[\texttt{authorized}\ N\ M];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[\texttt{throw}\ M'];\, \Theta', \Delta', \Sigma} \quad (\textit{AUTHROW1})$$

$$\frac{M;\, \Theta, \{\}, [] \;\overset{*}{\rightharpoonup}\; \texttt{return}\ M';\, \Theta', \Delta', \Sigma' \qquad \Sigma' \vdash N \overset{*}{\rightsquigarrow} \texttt{throw}\ N'}{\mathbb{S}[\texttt{authorized}\ N\ M];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[\texttt{throw UnathorizedError}];\Theta', \Delta', \Sigma} \quad (\textit{AURET2})$$

$$\frac{M;\Theta, \{\}, [] \;\overset{*}{\rightharpoonup}\; \texttt{throw}\ M';\, \Theta', \Delta', \Sigma' \qquad \Sigma' \vdash N \overset{*}{\rightsquigarrow} \texttt{throw}\ N'}{\mathbb{S}[\texttt{authorized}\ N\ M];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[\texttt{throw UnauthorizedError}];\, \Theta', \Delta', \Sigma} \quad (\textit{AUTHROW1})$$

$$\frac{M;\Theta, \{\}, [] \;\overset{*}{\rightharpoonup}\; \texttt{retry};\, \Theta', \Delta', \Sigma'}{\mathbb{S}[\texttt{authorized}\ N\ M];\, \Theta, \Delta, \Sigma \;\Rightarrow\; \mathbb{S}[\texttt{retry}];\, \Theta', \Delta'\Sigma} \quad (\textit{AURETRY})$$

Figure 2.4: Operational semantics for STM actions

$$\boxed{\text{TMI transitions} \qquad M;\, \Theta, \Delta, \Sigma \rightharpoonup N;\, \Theta', \Delta', \Sigma'}$$

$$\begin{array}{rcl}
\mathbb{S}[\texttt{readTMIVar}\ r];\, \Theta, \Delta, \Sigma & \rightharpoonup & \mathbb{S}[\texttt{return}\ \Theta_1(r)];\, \Theta, \Delta, \Sigma \oplus [(\textsc{read}, \Theta_2(r))] \\
& & \quad \text{if } r \in \mathrm{dom}(\Theta) \text{ and } \Theta_2(r)) \neq \bot \qquad \textit{(TMIREAD)} \\
\mathbb{S}[\texttt{writeTMIVar}\ r\ M];\, \Theta, \Delta, \Sigma & \rightharpoonup & \mathbb{S}[\texttt{return ()}];\, \Theta[r \mapsto (M, \Theta_2(r))], \Delta, \Sigma \oplus [(\textsc{write}, \Theta_2(r))] \\
& & \quad \text{if } r \in \mathrm{dom}(\Theta) \text{ and } \Theta_2(r)) \neq \bot \qquad \textit{(TMIWRITE)} \\
\mathbb{S}[\texttt{newTMIVar}\ N\ M];\, \Theta, \Delta, \Sigma & \rightharpoonup & \mathbb{S}[\texttt{return}\ r];\, \Theta[r \mapsto (M, N)], \Delta[r \mapsto (M, N)], \Sigma \oplus [(\textsc{create}, N)] \\
& & \quad r \notin \mathrm{dom}(\Theta) \qquad \textit{(TMINEW)}
\end{array}$$

$$\frac{M \to N}{\mathbb{S}[M];\, \Theta, \Delta, \Sigma \rightharpoonup \mathbb{S}[N];\, \Theta, \Delta, \Sigma} \quad \textit{(TADMIN)}$$

$$\frac{M;\, \Theta, \Delta, \Sigma \overset{*}{\Rightarrow} N;\, \Theta', \Delta', \Sigma'}{\mathbb{S}[\texttt{liftSTM}\ M];\, \Theta, \Delta, \Sigma \rightharpoonup \mathbb{S}[N];\, \Theta', \Delta', \Sigma'} \quad \textit{(LIFTSTM)}$$

$$\frac{M_1;\, \Theta, \Delta, \Sigma \overset{*}{\rightharpoonup} \texttt{return}\ N;\, \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1\ \texttt{'orElse'}\ M_2];\, \Theta, \Delta, \Sigma \rightharpoonup \mathbb{S}[\texttt{return}\ N];\, \Theta', \Delta', \Sigma'} \quad \textit{(TOR1)}$$

$$\frac{M_1;\, \Theta, \Delta, \Sigma \overset{*}{\rightharpoonup} \texttt{throw}\ N;\, \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1\ \texttt{'orElse'}\ M_2];\, \Theta, \Delta, \Sigma \rightharpoonup \mathbb{S}[\texttt{throw}\ N];\, \Theta', \Delta', \Sigma'} \quad \textit{(TOR2)}$$

$$\frac{M_1;\, \Theta, \Delta, \Sigma \overset{*}{\rightharpoonup} \texttt{retry};\, \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1\ \texttt{'orElse'}\ M_2];\, \Theta, \Delta, \Sigma \rightharpoonup \mathbb{S}[M_2];\, \Theta, \Delta, \Sigma} \quad \textit{(TOR3)}$$

$$\frac{M;\, \Theta, \{\}, [] \overset{*}{\rightharpoonup} \texttt{return}\ M';\, \Theta', \Delta', \Sigma'}{\mathbb{S}[\texttt{catch}\ M\ N];\, \Theta, \Delta, \Sigma \rightharpoonup \mathbb{S}[\texttt{return}\ M'];\, \Theta', \Delta \cup \Delta', \Sigma \oplus \Sigma'} \quad \textit{(XTMI1)}$$

$$\frac{M;\, \Theta, \{\}, [] \overset{*}{\rightharpoonup} \texttt{throw}\ M';\, \Theta', \Delta', \Sigma'}{\mathbb{S}[\texttt{catch}\ M\ N];\, \Theta, \Delta, \Sigma \rightharpoonup \mathbb{S}[N\ M'];\, \Theta \cup \Delta', \Delta \cup \Delta', \Sigma \oplus (\Sigma'|_{\Delta'})} \quad \textit{(XTMI2)}$$

$$\frac{M;\, \Theta, \{\}, [] \overset{*}{\rightharpoonup} \texttt{retry};\, \Theta', \Delta', \Sigma'}{\mathbb{S}[\texttt{catch}\ M\ N];\, \Theta, \Delta, \Sigma \rightharpoonup \mathbb{S}[\texttt{retry}];\, \Theta, \Delta, \Sigma} \quad \textit{(XTMI3)}$$

$$\boxed{\text{Authorization transitions} \qquad \Sigma \vdash M \rightsquigarrow N}$$

$$\frac{M \to N}{\Sigma \vdash \mathbb{E}[M] \rightsquigarrow \mathbb{E}[N]} \quad \textit{(AUADMIN)}$$

$$\Sigma \vdash \mathbb{E}[\texttt{getlog}] \rightsquigarrow \mathbb{E}[\texttt{return}\ hs(\Sigma)] \quad \textit{(GETLOG)}$$

$$\Sigma \vdash \mathbb{E}[\texttt{catch (return}\ M\texttt{)}\ N] \rightsquigarrow \mathbb{E}[\texttt{return}\ M] \quad \textit{(ACATCH1)}$$

$$\Sigma \vdash \mathbb{E}[\texttt{catch (throw}\ M\texttt{)}\ N] \rightsquigarrow \mathbb{E}[N\ M] \quad \textit{(ACATCH2)}$$

Figure 2.5: Operational semantics for TMI actions

are transaction local, i.e. they are reset at the start of each atomic sequence of reductions in the STM system, see e.g. rule *(ARET)*.

The first three rules define actions on transactional variables. They operate on the store Θ but only on those variables where the second component (i.e. the security descriptor) is $\perp$. This is what differentiates regular transactional variables from security sensitive variables.

Other rules in the STM transition system define the behavior of STM combinators as described in Section 2.2.2. We have used the revised semantics of exception handling from the later versions of [Harris et al. 2005], namely the rule *XTM2*, to ensure that when a term reduction results in a caught exception, all effects of that reduction are rolled back except for new allocations.

TMI transitions: A key TMI addition to the STM transitions is the handling of `authorized`. The rules *AURETn* and *AUTHROWn* specify that for a term of the form (`authorized` N M) if M evaluates to a **`return`** or `throw` value, then that value is propagated only if N evaluates to a **`return`** value under the authorization relation (see later). If evaluation of the *authorization term N* raises an exception, [2] a fixed exception containing no information about the local transaction state is thrown. This triggers a rollback of any updates performed during that invocation of `authorized`.

We should note that in the case of an exception, including authorization failure, new allocations are retained for the reason described above. In the implementation, this is not done explicitly as deallocation of references is handled by the garbage collector. Any new allocations that are actually referenced by exception values are thus retained, but others are discarded. Thus, since we don't allow any variable references in our special exception for authorization failures, no new allocations will leak in practice.

Figure 2.5 shows the two new TMI transition relations. The first one deals with TMI actions and is indicated by the symbol $\rightharpoonup$. The configurations of this transition system are identical to those of the STM system. Indeed, TMI actions behave very much like STM actions, the main difference is that variable operations in TMI actions can operate on security sensitive variables. A variable v is security sensitive if and only if $\Theta_2(v) \neq \perp$. The first three rules of Figure 2.5 describe the variable operations. When these operations are performed, a log entry is added to Σ with the contents of the variable's security descriptor. Another addition over STM behavior is rule *LIFTSTM*. This rule states that any sequence of STM reductions can be lifted to the TMI level. This is necessary to allow TMI code to access regular transactional variables, and should be possible, since TMI actions are always performed in the context of an enclosing STM action.

Finally, we add a separate transition system to evaluate authorization functions. In this system a transition of the form

$$\Sigma \vdash M \rightsquigarrow N$$

[2] In the actual implementation the authorization term is simply a boolean function of the log. This difference exists to simplify both the semantics and the implementation.

represents the reduction of term M to term N, under the context of an introspection log Σ. The reason for this notation is that the introspection log is fixed, i.e. read-only, for these transitions. This system only allows pure operations and monad binding via the administrative transitions, the usual exception handling and one special term `getlog`. The `getlog` term is reduced to a list representation (in the Haskell sense) of the access log Σ. The terms reduced with this system can thus examine the log and make decisions based on its contents.

An example: As an example of reading and applying the rules, consider the program

```
atomically
  (
     authorized (assert (isEmpty getlog))
                (writeTMIVar x 10)
  )
```

Working from the inside out, we can see that the innermost expression of `writeTMIVar x 10` will update the value of x in Θ as well as enter an entry to the introspection log Σ, by applying rule *(TMIWRITE)*. The resulting term is **`return`** `()`. As the resulting log is non-empty, the authorization term `assert (isEmpty getlog)` will throw an exception. Thus, for the `authorized` term, rule *(AURET2)* is the only applicable one, so that term evaluates to `throw UnauthorizedError`. The `atomically` term is therefore evaluated to the same result via rule *(ATHROW)*, but this rule does not preserve updates to the store Θ, meaning that the transaction has been aborted.

Nested TMI actions: As we mentioned in the previous section, the capability of lifting STM actions up to the TMI levels allows us to nest TMI actions. An inner TMI action can be authorized with a separate authorization term. Consider the following example of an action that provides a student with information about her grade for a course, as well as the average of all grades of other students. Naturally, the student doesn't have access to other students' grades but for the purpose of calculating the average we may allow such access in a nested action.

Assume that we have defined the following terms.

- `ownGradesRead s` is an authorization term that succeeds only if the input log only contains reading of grades that belong to student `s`
- `allGradesRead` is an authorization term that succeeds only if the log only contains reading of grades, but regardless of the owner of the grades. This may be considered a kind of a *system* read access to grades.
- `readGrade s` is a TMI action that reads a grade of a student from the relevant `TMIVar` and returns it. The introspection log will contain an appropriate entry afterwards.
- `averageGrades` is a TMI action that reads grades of all students from the appropriate `TMIVars` and returns their average. The introspection log will contain an entry for every read grade.

Now it is possible to define the following TMI action that provides a student with her own grade as well as the average grades of all students.

```
gradeInformation :: Student -> TMI (Grade, Grade)
gradeInformation s =
  do own <- readGrade s
     avg <- liftSTM (authorized allGradesRead averageGrades)
     return (own, avg)
```

This function may be called with the appropriate authorization function, namely one that only allows a student access to her own grades.

```
atomically (authorized (ownGradesRead s) (gradeInformation s))
```

By applying the operational semantics rules to this term, one can find that the innermost action averageGrades will be authorized by allGradesRead *before* turning it into an STM transition. This may, for example, happen through rule *(AURET1)*. Note that such a rule does not keep the log entries of already authorized actions, i.e. the Σ is not affected in the $\Rightarrow$ transition below the line. Thus the log of the nested action is not contained in the log authorized by the outer authorization function ownGradesRead.

This use of nesting constitutes a *privilege amplification* in a manner similar to stack inspection [Fournet and Gordon 2003].

2.5 Implementation

Our Haskell implementation is comprised of one module, TMI. The most important components are the monad TMI d a and the type for TMI variables, TMIVar d a. Both are parameterized on the descriptor type d, which is chosen by the user of this module. A TMI variable is represented by a descriptor value and an STM TVar,

```
data TMIVar d a = TMIVar {
      getTVar       :: TVar a,
      getDescriptor :: d
}
```

The field accessors getTVar and getDescriptor are not exported and only available inside the TMI module.

The TMI monad is a stack of the standard *writer monad* on top of the regular STM monad. The writer monad has the ability of collecting accumulating information in a sequence, which is exactly what we need to maintain the introspection log. In the TMI module, the regular STM module is imported under the name T.

```
newtype TMI d a = TM {
   unwrapTM :: WriterT (TMILog d) T.STM a
} deriving (Monad)
```

The type TMILog d represents a log of all accesses to TMI variables. It is defined by the following declarations.

```
data TMIAccess = CreateVar | ReadVar | WriteVar
type TMILog d = [(TMIAccess, d)]
```

For inserting entries in the log, we define the following shortcut, where `tell` is the standard function that the writer monad uses to collect information.

```
log :: (TMIAccess, d) -> TMI d ()
log entry = (TM . tell) [entry]
```

`log` returns an STM action which has the only effect of appending its argument to the introspection log. Note that this helper is not exported, so users of the TMI module cannot append to the log directly. As expected, a log entry of the form `(ReadVar, x)` just means that a variable with descriptor value `x` was read.

The `liftSTM` function lifts an STM operation to a TMI operation. The log is not affected by the work performed in the STM action.

```
liftSTM :: STM a -> TMI d a
liftSTM = TM . lift
```

The functions to create, read and write TMI variables are now simple to define. They all enter the relevant entries to the log and then call the underlying functions from the STM module.

```
newTMIVar :: d -> a -> TMI d (TMIVar d a)
newTMIVar description val =
    do log (CreateVar, description)
       var <- liftSTM (T.newTVar val)
       return (TMIVar var description)

readTMIVar :: TMIVar d a -> TMI d a
readTMIVar tv =
    do log (ReadVar, getDescriptor tv)
       liftSTM (T.readTVar (getTVar tv))

writeTMIVar :: TMIVar d a -> a -> TMI d ()
writeTMIVar tv val =
    do log (WriteVar, getDescriptor tv)
       liftSTM (T.writeTVar (getTVar tv) val)
```

The combinators from the STM world are defined thus.

```
retryTMI = liftSTM T.retry

runTMI = runWriterT . unwrapTM   -- helper

orElseTMI t1 t2 = TM . WriterT (runTMI t1 `T.orElse` runTMI t2)
```

What is left is to define the crucial `authorized` function. This function accepts an authorization function with the type `TMILog d -> Bool` and a TMI action; it should return an STM action that performs the operation of the TMI action, validates the resulting log with the authorization function and either returns the result or throws an exception. With this description in mind, the implementation is pretty straight-forward.

```
authorized :: (TMILog d -> Bool) -> TMI d a -> T.STM a
authorized auth act =  do
   (result, log) <- runTMI act
   if not (auth log)
      then throw (AssertionFailed "Access␣denied")
      else return result
```

Note that a custom exception may be more appropriate but for sake of clarity we simply use the standard assertion failure to trigger a transaction abort.

Our Haskell TMI implementation can support variants of history-based policy enforcement [Abadi and Fournet 2003], in particular allowing *privilege amplification* with nested TMI actions as described in the previous section. In particular a call to `authorized` will return an STM action which *contains* a TMI operation and an associated authorization closure. When code inside a TMI action needs increased privileges it can nest a call to `authorized` with a different authorization manager and use `liftSTM` to lift the resulting STM operation back to the TMI level.

As in stack inspection [Fournet and Gordon 2003], privilege amplification provides TMI security managers with a useful escape hatch to perform operations as a more powerful ‘‘application principal’’.

2.6 Discussion and future work

We have presented both a formal semantics and an implementation of TMI over the Haskell STM system. During this work, we discovered that there are many design decisions to be made and the design we have presented here is only one of many possibilities. The variants we experimented with in the design process did not always exhibit the behaviour that we expected or wanted. For this task, defining the formal semantics proved to be an essential tool to understand and evaluate different design decisions as well as spotting special cases that were not so obvious in the actual implementation. Indeed, constructing the semantics helped us discover bugs and unexpected behaviour in the code, even after weeks of careful consideration. In addition the formal semantics gives a clear and unambiguous description of the TMI architecture.

The TMI architecture, as described in Section 2.3.1, can support the enforcement of stateful security policies that depend on the execution history over multiple transactions. In our paper [Birgisson et al. 2008], we have experimented with such policies in another implementation of the TMI architecture. However, we have not explored the addition of such facilities here, in order to simplify the exposition of our semantics and Haskell implementation.

The privilege amplification by nested TMI actions naturally relies on the programmer to ensure that the nested authorization manager does not violate the enclosing policy. The objective of the TMI architecture is to provide facilities for writing policy enforcement code, not to prevent injection of malicious code. In the case of library development where one deals with untrusted code, such nesting may not be desirable and can be disabled. We have experimented with other ways of implementing privilege amplification without relying on this nesting, with good results.

In our implementation we are maintaining the introspection log by hand. This works well for prototypical purposes, but we would like to investigate the possibility of making use of the real underlying transaction log. This requires modifications to the STM framework provided in the GHC runtime library. Having the formal semantics as the definition of the desired behaviour should make such an implementation easier to construct and check.

Future work in this context also involves writing or porting complex software to the architecture to obtain realistic performance measurements. Also, our semantics may still be simplified, while allowing the same behavior; for example, the transaction log seems redundant in STM transitions, and may possibly be eliminated.

Most importantly, we think that having clear semantics for TMI and an implementation over a production-ready STM system, further validates our claim that TMI architecture is very relevant to practical software development.

chapter three

Decompositional Reasoning about the History of Parallel Processes

joint work with Luca Aceto, Anna Ingólfsdottir, and MohammadReza Mousavi

3.1 Introduction

State-space explosion is a major obstacle in model-checking logical properties. One approach to combat this problem is compositional reasoning, where properties of a system as a whole are deduced from the properties of its components. Decompositional reasoning [Giannakopoulou et al. 2005, Xie and Dang 2006, Andersen 1995, Laroussinie and Larsen 1995, Larsen and Xinxin 1991] often improves upon compositional reasoning by automatically decomposing the global property to be model checked into local properties of (possibly unknown) components. For example, Andersen's paper shows the effectiveness of the method in the analysis of Milner's scheduler [Milner 1989]. In the context of process algebras, as the specification language, and Hennessy-Milner logic, as the logical formalism for properties, decompositional reasoning techniques date back to the seminal work of Larsen and Xinxin in the 80's and early 90's [Larsen and Xinxin 1991], which is further developed in [Ingólfsdóttir et al. 1987, Simpson 2004, Fokkink et al. 2006]. However, we are not aware of any such decomposition technique which applies to reasoning about the "past". This is particularly interesting in the light of recent developments concerning reversible processes [Phillips and Ulidowski 2006] and knowledge representation (epistemic aspects) inside process algebra [Dechesne et al. 2007, Borgström et al. 2006, Hommersom et al. 2004, Raimondi and Lomuscio 2007, Halpern and O'Neill 2005], all of which involve some notion of specification and reasoning about the past.

In this chapter, we tackle this problem and present a decomposition technique for Hennesy–Milner logic with past. As the specification language, we use a subset CCS with parallel composition, non-deterministic choice, action prefixing and the inaction constant. The rest of the chapter is structured as follows. Section 3.2 introduces preliminary definitions and the extension of Hennessy–Milner logic with past. Section 3.3 discusses how parallel computations that maintain their history are decomposed into their parallel components. Section 3.4 presents the decompositional reasoning and the main theorem of the chapter, and Section 3.5 discusses related work and possible extensions to our results.

3.2 Preliminaries

3.2.1 Computations and CCS

The following definitions come mostly from [Nicola et al. 1990].

Definition 3.1. *(Labelled transition system)* A *labelled transition system* (LTS) is a triple $\langle P, A, \longrightarrow \rangle$ where

- P is a set of process names.
- A is a finite set of action names, not including a *silent action* τ. We write A_τ for $A \cup \{\tau\}$.
- $\longrightarrow \subseteq P \times A_\tau \times P$ is the *transition relation*, we call its elements *transitions* and usually write $p \xrightarrow{\alpha} p'$ to mean that $(p, \alpha, p') \in \longrightarrow$.

We let $p, q, \ldots$ range over P, $a, b, \ldots$ over A and $\alpha, \beta, \ldots$ over A_τ.

Definition 3.2. *(Sequences and computations)* For any set S we let S^* be the set of finite sequences of elements from S. Concatenation of sequences is represented by juxtaposition. λ denotes the empty sequence and $|\sigma|$ stands for the length of a sequence σ.

Given an LTS $\mathcal{T} = \langle P, A, \longrightarrow \rangle$, we define a *path from* p_0 to be a sequence of transitions

$$p_0 \xrightarrow{\alpha_0} p_1, p_1 \xrightarrow{\alpha_1} p_2, \ldots, p_n \xrightarrow{\alpha_{n-1}} p_n$$

and usually write this as

$$p_0 \xrightarrow{\alpha_0} p_1 \xrightarrow{\alpha_1} p_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} p_n.$$

We use $\pi, \mu, \ldots$ to range over paths. A *computation from* p is a pair (p, π) where π is a path from p and we use $\rho, \sigma, \ldots$ to range over computations. $\mathcal{C}_\mathcal{T}(p)$, or simply $\mathcal{C}(p)$ when the LTS $\mathcal{T}$ is clear from the context, is the set of computations from p and $\mathcal{C}_\mathcal{T}$ is the set of all computations in $\mathcal{T}$.

For a computation $\rho = (p_0, \pi)$ where $\pi = p_0 \xrightarrow{\alpha_0} p_1 \xrightarrow{\alpha_1} p_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} p_n$ we define

$$\begin{aligned}
\text{first}(\rho) &= \text{first}(\pi) = p_0 \\
\text{last}(\rho) &= \text{last}(\pi) = p_n \\
\text{trace}(\rho) &= (\alpha_0 \ldots \alpha_{n-1}) \in A_\tau^* \\
|\rho| &= |\pi| = n
\end{aligned}$$

We refer to elements of A^* as *traces*.

Concatenation of computations ρ and ρ' is denoted by their juxtaposition $\rho\rho'$ and is defined iff $\mathrm{last}(\rho) = \mathrm{first}(\rho')$. When $\mathrm{last}(\rho) = p$ we will write $\rho(p \xrightarrow{\alpha} q)$ as a shorthand for the slightly longer $\rho(p, p \xrightarrow{\alpha} q)$. We also use $\rho \xrightarrow{\alpha} \rho'$ to denote that there exists a computation $\sigma = (p, p \xrightarrow{\alpha} p')$, for some processes p and p', such that $\rho' = \rho\sigma$.

Remark. Representing computations with a pair (p, π) might seem redundant at first, since π must start with p. However, an empty computation (p, λ) is also valid and must be distinguished from (q, λ) if $p \neq q$.

3.2.2 Hennessy-Milner Logic with Past

Definition 3.3. *(Hennessy Milner logic with past)* Let $\mathcal{T} = \langle P, A, \rightarrow\rangle$ be an LTS. The set $HML_{\leftarrow}(A)$, or simply $HML_{\leftarrow}$, of *Hennessy-Milner logic formulae with past* is defined by the following grammar, where $\alpha \in A_\tau$.

$$\varphi, \psi ::= \top \mid \varphi \wedge \psi \mid \neg\varphi \mid \langle\alpha\rangle\varphi \mid \langle\leftarrow\alpha\rangle\varphi.$$

We define the *satisfaction relation* $\vDash \subseteq \mathcal{C}_\mathcal{T} \times HML_{\leftarrow}$ as the least relation that satisfies the following clauses:

- $\rho \vDash \top$ for all $\rho \in \mathcal{C}_\mathcal{T}$,
- $\rho \vDash \varphi \wedge \psi$ iff $\rho \vDash \varphi$ and $\rho \vDash \psi$,
- $\rho \vDash \neg\varphi$ iff not $\rho \vDash \varphi$,
- $\rho \vDash \langle\alpha\rangle\varphi$ iff $\rho \xrightarrow{\alpha} \rho'$ and $\rho' \vDash \varphi$ for some $\rho' \in \mathcal{C}_\mathcal{T}$,
- $\rho \vDash \langle\leftarrow\alpha\rangle\varphi$ iff $\rho' \xrightarrow{\alpha} \rho$ and $\rho' \vDash \varphi$ for some $\rho' \in \mathcal{C}_\mathcal{T}$.

The satisfaction relation $\vDash \subseteq P \times HML_{\leftarrow}$ is defined by $p \vDash \varphi$ if and only if $(p, \lambda) \vDash \varphi$.

We will make use of some standard shorthands for Hennessy-Milner type logics.

$$\begin{aligned}\bot &= \neg\top\\ \varphi \vee \psi &= \neg(\neg\varphi \wedge \neg\psi)\\ [\alpha]\varphi &= \neg\langle\alpha\rangle(\neg\varphi)\\ [\leftarrow\alpha]\varphi &= \neg\langle\leftarrow\alpha\rangle(\neg\varphi)\end{aligned}$$

For a finite set of actions A, we will also use the following notation.

$$\langle A\rangle\varphi = \bigvee_{\alpha\in A}\langle\alpha\rangle\varphi \qquad \langle\leftarrow A\rangle\varphi = \bigvee_{\alpha\in A}\langle\leftarrow\alpha\rangle\varphi$$

$$[A]\varphi = \bigwedge_{\alpha\in A}[\alpha]\varphi \qquad [\leftarrow A]\varphi = \bigwedge_{\alpha\in A}[\leftarrow\alpha]\varphi$$

Intuitively, $\rho \vDash \langle \leftarrow \alpha \rangle \varphi$ means the last action of ρ *is* labelled with α and the preceding computation satisifies φ, while $\rho \vDash [\leftarrow \alpha]\varphi$ means that *if* the last computation is labelled with α, then the preceding computation satisfies φ. Specifically, a computation with an empty path, i.e. the initial state, can never satisfy the former while it will always vacuously satisfy the latter.

Determinism of the past

It is worth mentioning that the operators $\langle \cdot \rangle$ and $\langle \leftarrow \cdot \rangle$ are not entirely symmetric. The future is non-deterministic, i.e. at any point we may have a choice of multiple ways for a computation to proceed. In our semantics for *HML*$_{\leftarrow}$, the past however is always deterministic; there is always at most one transition that was the last transition to occur. This is by design, and we could have chosen to model the past as nondeterministic as well, i.e. to take a possibilistic view where we would consider all possible histories.

This would make the forward and backwards diamond operators symmetric in their view of the process graph. However, we are more interested in properties about the actual past of a computation, especially w.r.t. modelling epistemic properties, which rely on observations of some aspects of the computation so far. (We do not discuss such properties in the current chapter.) We have thus reached the same conclusion as [Laroussinie and Schnoebelen 2000] that the deterministic view is more appropriate for our purposes. Laroussinie and Schnoebelen list two other properties of their model of the past in addition to being deterministic, namely that the past is finite (there is a fixed initial state) and that past is cumulative (at each transition the history gains information). We make implicit use of the latter property in our proofs and find that the former is a natural property of the processes we want to model.

Treating the past as deterministic might suggest that it is unnecessary to index the operator $\langle \leftarrow \alpha \rangle$ with the action and instead provide an operator $\langle \leftarrow \rangle$ which matches any action (since there is at most one). However this turns out to be insufficient, as such a logic could not distinguish between the two computations

$$(r, r \xrightarrow{\alpha} t) \quad \text{and} \quad (r, r \xrightarrow{\beta} s)$$

when $\alpha \neq \beta$. For our intended purposes, it is important that we are able to tell these two apart.

3.3 Decomposing Computations

We seek a definition of a *formula quotient w.r.t. a process/state* following the work of [Larsen and Xinxin 1991]. This might be written as φ/p where φ is an *HML*$_{\leftarrow}$ formula and p is a process. The theorem we then seek to prove is this:

$$p \parallel q \vDash \varphi \quad \Leftrightarrow \quad p \vDash \varphi/q$$

where the *parallel composition operator* $\parallel$ is suitably defined over LTSs.

Given our definition of $\models$ for $HML_{\leftarrow}$, this requires us to prove a theorem of the form

$$\rho \models \varphi \quad \Leftrightarrow \quad \rho_1 \models \varphi/\rho_2$$

where ρ, ρ_1, ρ_2 are computations such that ρ is a computation of a process of the form $p \parallel q$ and that is, in some sense, the "parallel composition" of ρ_1 and ρ_2. In the standard setting it is straightforward to identify the components of a parallel composition. In the case of computations, however, this is not so obvious. A computation composed of two processes run in parallel has the form

$$(p \parallel q, \pi)$$

where $p \parallel q$ is a syntactic representation of the initial state and π is the path leading up to the current state. The path π however may involve contributions from both of the parallel components. Separating the contributions of the components for the purposes of decompositional model checking requires us to *unzip* these paths into separate paths that might have been observed by considering only one argument of the composition. This means that we have to find two paths π_p and π_q such that the computations

$$(p, \pi_p) \quad \text{and} \quad (q, \pi_q)$$

are in some sense independent computations that run in parallel will yield $(p \parallel q, \pi)$.

3.3.1 Decomposition of computations

In the setting of HML without past, parallel composition may be defined directly on LTSs independent of the syntax or semantics of the underlying process algebra. When dealing with computations, this does not provide enough information to find the two computations that make up the parallel composition. For this information, one needs to look into the syntax and semantics of the processes themselves and moreover their semantics have to follow some restrictions.

For this study, in order to highlight the main ideas and technical tools in our approach, we will restrict ourselves to a subset of CCS, namely CCS without renaming, restriction or recursion. (We will discuss possible extensions of our results in Section 3.5.) Processes are thus defined by the following grammar.

$$p, q \quad ::= \quad 0 \mid \alpha.p \mid p + q \mid p \parallel q$$

with the following operational semantics

$$\frac{}{\alpha.p \xrightarrow{\alpha} p} \qquad \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{a} q'}$$

$$\frac{p \xrightarrow{\alpha} p'}{p \parallel q \xrightarrow{\alpha} p' \parallel q} \qquad \frac{q \xrightarrow{\alpha} q'}{p \parallel q \xrightarrow{\alpha} p \parallel q'} \qquad \frac{p \xrightarrow{a} p' \quad q \xrightarrow{\bar{a}} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'}$$

We write $p \xrightarrow{\alpha} q$ to denote that this transition is provable by these rules. We assume also that $\bar{\cdot} : A \to A$ is a function on action names such that $\bar{\bar{a}} = a$. The LTS associated with a CCS process p is the largest transition system generated by these rules starting from the process p.

The decomposition of a computation running two parallel components must retain the information about the order of steps in the interleaved computation. We do this by modelling the decomposition using *stuttering computations*. These are computations that are not only sequences of transition triplets, but may also involve pseudo steps labelled with $\dashrightarrow$. Intuitively, $p \dashrightarrow p$ means that process p has remained idle in the last transition performed by a parallel process having p as one of its parallel components. We denote the set of stuttering computations with $\mathcal{C}^*_{\mathcal{T}}$ or simply $\mathcal{C}^*$. For example, the computation

$$(a.0 \parallel b.0, a.0 \parallel b.0 \xrightarrow{a} 0 \parallel b.0 \xrightarrow{b} 0 \parallel 0)$$

is decomposed into the stuttering computations

$$(a.0, a.0 \xrightarrow{a} 0 \dashrightarrow 0) \quad \text{and}$$
$$(b.0, b.0 \dashrightarrow b.0 \xrightarrow{b} 0).$$

However, the decomposition of a parallel computation is not in general unique, as there may be several possibilities stemming from different synchronisation patterns. For example consider a computation with the following trace.

$$(a.0 + b.0) \parallel (\bar{a}.0 + \bar{b}.0) \xrightarrow{\tau} 0 \parallel 0$$

From this computation it is not possible to distinguish if the transition labelled with τ was the result of communication of the a and $\bar{a}$ actions, or of the b and $\bar{b}$ actions. For our purposes, this is not necessarily a problem because no expression of our logic can differentiate between the two synchronisations, given only the composed computation. We thus consider all possibilities simultaneously, i.e. a decomposition of a computation will actually be *a set* of pairs of components.

The following function over paths defines the decomposition of a computation.

$$D(\lambda) = \{(\lambda, \lambda)\}$$
$$D(\pi'(p' \parallel q' \dashrightarrow p' \parallel q')) = \{(\mu_1(p' \dashrightarrow p'), \mu_2(q' \dashrightarrow q')) \mid (\mu_1, \mu_2) \in D(\pi')\}$$
$$D(\pi'(p' \parallel q' \xrightarrow{\alpha} p'' \parallel q'')) = \begin{cases} \{(\mu_1(p' \xrightarrow{\alpha} p''), \mu_2(q' \dashrightarrow q')) \\ \qquad \mid (\mu_1, \mu_2) \in D(\pi')\} & \text{if } q' \equiv q'' \\ \{(\mu_1(p' \dashrightarrow p'), \mu_2(q' \xrightarrow{\alpha} q'')) \\ \qquad \mid (\mu_1, \mu_2) \in D(\pi')\} & \text{if } p' \equiv p'' \\ \{(\mu_1(p' \xrightarrow{a} p''), \mu_2(q' \xrightarrow{\bar{a}} q'')) \\ \qquad \mid (\mu_1, \mu_2) \in D(\pi'), a \in A, \\ \qquad\quad p' \xrightarrow{a} p'', q' \xrightarrow{\bar{a}} q''\} & \text{otherwise and } \alpha = \tau \end{cases}$$

We should make a note of the fact that if (μ_1, μ_2) is a decomposition of a computation π, then the three computations have the same length. Furthermore

$$\text{last}(\pi) = \text{last}(\mu_1) \parallel \text{last}(\mu_2). \tag{3.1}$$

Also of interest is that, even though the above definition yields a set of decompositions of π, the only case where multiple possibilities are generated is the last case where both components evolve, and where there is ambiguity in the processes as to which actions actually contributed to the communication. As we mentioned above, there is no expression of the $HML_{\leftarrow}$ logic that can resolve such ambiguity only by looking at a composed computation. Since our goal is to model check of such expressions, the existence of multiple decompositions of one computation will not pose any problem.

Another notable property of path decomposition, is that its inverse is unique, i.e. a pair (μ_1, μ_2) can only be the decomposition of a single path. We formalise this as a lemma which will come in handy later.

Lemma 3.4. Let π_1 be a path of a parallel computation and $(\mu_1, \mu_2) \in D(\pi_1)$. If π_2 is a path such that $(\mu_1, \mu_2) \in D(\pi_2)$ also, then $\pi_1 = \pi_2$.

Proof. We start by noting that π_1 and π_2 cannot differ in length, as they are both equal in length to μ_1 (and μ_2). We apply induction on their common length.

If both are empty, $\pi_1 = \pi_2 = \lambda$, then there is nothing to prove. Now assume they are non-empty and that

$$\begin{aligned} \pi_1 &= \pi_1'(p_1' \parallel q_1' \; R_1 \; p_1 \parallel q_1) \\ \pi_2 &= \pi_2'(p_2' \parallel q_2' \; R_2 \; p_2 \parallel q_2) \end{aligned}$$

where R_1, R_2 are relations of the form $\xrightarrow{\alpha}$ or $\dashrightarrow$. The induction hypothesis states that $\pi_1' = \pi_2'$, which also means that $p_1' = p_2'$ and $q_1' = q_2'$. Property (3.1) above furthermore gives that $p_1 = p_2$ and $q_1 = q_2$. Thus we only need to show that the final steps coincide also, i.e. that $R_1 = R_2$. The proof proceeds by case analysis on the last steps of μ_1 and μ_2.

- If both μ_1 and μ_2 end with a pseudo-step, then we see from the definition of D that both R_1 and R_2 must be pseudo-transitions.

- If only one of μ_1 and μ_2 ends with a pseudo-step, then the action of the other one must be the same as the last action of both π and π'.

- If both μ_1 and μ_2 end with a proper transition, we note that by the definition of D the actions must complement each other. Then the last step of both π and π' must thus be labelled with τ.

- If both μ_1 and μ_2 end with a proper transition, we note that by the definition of D the actions must complement each other. Then the last step of both π and π' must thus be labelled with τ.

This covers all the cases and thus we have shown that $R_1 = R_2$, $p_1 = p_2$ and $q_1 = q_2$. Coupled with the induction hypothesis, this means that $\pi = \pi'$. □

Note that the definition of D relies on some properties of CCS specifically.

1. We must have that $p \xrightarrow{a} p'$ leads to $p \not\equiv p'$. This is necessary so that the case-definitions are well defined, i.e. that they are mutually exclusive. This means that we can rely on $\equiv$-testing to determine if one side of the composition took a step or not.

   We should also note that this requirement means that we can actually remove the text "and $\alpha = \tau$" from the last case condition. To see why, note that the condition $q' \not\equiv q'' \wedge p' \not\equiv p''$ (as implied by the word "otherwise") means that both must have taken a step simultaneously and communicated, and therefore the only possible result action is indeed τ. This means that the definition properly covers all cases.

2. The only possible result of a communication is τ, and τ can never act as one partner of the communication.

We now want to define quotient on $HML_{\leftarrow}$-formulae such that a property of the form

$$(p \parallel q, \pi) \models \varphi \quad \Leftrightarrow \quad (p, \mu_1) \models \varphi/(q, \mu_2)$$

where $(\mu_1, \mu_2) \in D(\pi)$. However, since we are dealing with sets of decompositions, we need to quantify over these sets. It turns out that a natural way that also gives a strong result is the following. Given that a composed computation satisfies a formula, we can prove that one component of *every* decomposition satisfies a formula quotiented with the other component,

$$(p \parallel q, \pi) \models \varphi \quad \Rightarrow \quad \forall (\mu_1, \mu_2) \in D(\pi) : (p, \mu_1) \models \varphi/(q, \mu_2).$$

On the other hand, to show the other direction, we need only one witness of a decomposition that satisfies a quotiented formula to deduce that the composed computation satisfies the original one,

$$\exists (\mu_1, \mu_2) \in D(\pi) : (p, \mu_1) \models \varphi/(q, \mu_2) \quad \Rightarrow \quad (p \parallel q, \pi) \models \varphi.$$

Before defining the quotienting transformation we need define what $\models$ means with respect to stuttering computations. We do this by extending $HML_{\leftarrow}$ to $HML^*_{\leftarrow}$ by adding two operators.

Definition 3.5. *(Stuttering Hennessy Milner logic with past)* Let $\mathcal{T} = \langle P, A, \rightarrow \rangle$ be an LTS. The set $HML^*_{\leftarrow}(A)$, or simply $HML^*_{\leftarrow}$, of *stuttering Hennessy-Milner logic formulae with past* is defined by the grammar

$$\varphi, \psi ::= \top \mid \varphi \wedge \psi \mid \neg\varphi \mid \langle \alpha \rangle \varphi \mid \langle \leftarrow \alpha \rangle \varphi \mid \langle \dashrightarrow \rangle \varphi \mid \langle \dashleftarrow \rangle \varphi$$

where $i \in \mathbb{N}$ and $\alpha \in A_\tau$. We define the *satisfaction relation* $\models^* \subseteq \mathcal{C}^*_{\mathcal{T}} \times HML^*_{\leftarrow}$ as the least relation that satisfies the following,

- $\rho \models^* \top$ for all $\rho \in \mathcal{C}^*_{\mathcal{T}}$,
- $\rho \models^* \varphi \wedge \psi$ iff $\rho \models^* \varphi$ and $\rho \models^* \psi$,
- $\rho \models^* \neg\varphi$ iff not $\rho \models^* \varphi$,

- $\rho \models^* \langle\alpha\rangle\varphi$ iff for some $\rho' \in \mathcal{C}^*_{\mathcal{T}} : \rho \xrightarrow{\alpha} \rho'$ and $\rho' \models^* \varphi$
- $\rho \models^* \langle\leftarrow\alpha\rangle\varphi$ iff for some $\rho' \in \mathcal{C}^*_{\mathcal{T}} : \rho' \xrightarrow{\alpha} \rho$ and $\rho' \models^* \varphi$
- $\rho \models^* \langle\dashrightarrow\rangle\varphi$ iff $\rho(p \dashrightarrow p) \models^* \varphi$ where $p = \text{last}(\rho)$.
- $\rho \models^* \langle\dashleftarrow\rangle\varphi$ iff $\rho' \models^* \varphi$ where $\rho = \rho'(p \dashrightarrow p)$ for some p.

The satisfaction relation $\models^* \in P \times HML^*_{\leftarrow}$ is defined by $p \models^* \varphi$ if and only if $(p, \lambda) \models^* \varphi$.

Remark. The satisfaction relations $\models^*$ and $\models$ coincide over $\mathcal{C}_{\mathcal{T}} \times HML_{\leftarrow}$.

3.3.2 Why are the stutters necessary?

One may ask why we need to extend both computations and the logic to include the notion of stuttering steps. The reason for doing so is to capture the information about the interleaving order in component computations. This in turn is necessary because the original logic can differentiate between different interleavings of parallel processes.

For an example, let p be a process that cannot perform an a action, but $p \xrightarrow{b} p'$ for some p'. Consider the computation $(a.0 \parallel p, \pi)$ where

$$\pi = a.0 \parallel p \xrightarrow{a} 0 \parallel p \xrightarrow{b} 0 \parallel p' \tag{3.2}$$

Clearly this computation does **not** satisfy the formula $\langle\leftarrow a\rangle\top$.

Another interleaving of the same parallel composition is the computation $(a.0 \parallel p, \pi')$ where

$$\pi' = a.0 \parallel p \xrightarrow{b} a.0 \parallel p' \xrightarrow{a} 0 \parallel p'. \tag{3.3}$$

This computation however does satisfy $\langle\leftarrow a\rangle\top$. Since the logic can distinguish between different interleaving orders of a parallel computation, it is vital to maintain information about interleaving order in our decomposition. If the decomposition of the above computations only consider the actions contributed by each component, this information is lost and both decompose to the same pair of computations and we cannot reasonably expect to test if they satisfy the formula $\langle\leftarrow a\rangle\top$ in a decompositional manner.

3.4 Decompositional Reasoning

We now define the quotienting construction over formulae structurally. Quotienting distributes over the boolean operators.

$$\begin{aligned}
\top/\rho &= \top \\
(\varphi_1 \wedge \varphi_2)/\rho &= \varphi_1/\rho \wedge \varphi_2/\rho \\
(\neg\varphi)/\rho &= \neg(\varphi/\rho)
\end{aligned}$$

The modal operators however need more attention. We start with $\langle\alpha\rangle\varphi$ and consider separately the cases where $\alpha \in A$ and $\alpha = \tau$. In the following we assume $p' = \text{last}(\rho)$.

$$(\langle a\rangle\varphi)/\rho = \langle a\rangle\,(\varphi/\rho(p' \dashrightarrow p')) \vee \left(\bigvee_{\rho':\rho \xrightarrow{a} \rho'} \langle\dashrightarrow\rangle(\varphi/\rho')\right)$$

$$(\langle\tau\rangle\varphi)/\rho = \langle\tau\rangle\,(\varphi/\rho(p' \dashrightarrow p')) \vee \left(\bigvee_{\rho':\rho \xrightarrow{\tau} \rho'} \langle\dashrightarrow\rangle(\varphi/\rho')\right) \vee \left(\bigvee_{\rho',a:\rho \xrightarrow{a} \rho'} \langle\bar{a}\rangle(\varphi/\rho')\right)$$

Intuitively, the first case states that when we expect the composed computation to be able to perform an a-transition, there are two possibilities. The first possibility is that the component we intend to test with the quotient formula can perform an a-transition. The rest of the formula must then be quotiented with ρ plus a pseudo-step representing that this component remained idle. The second possibility is that there is an a-transition from ρ. In this case the component we want to test must proceed with a pseudo-step. The same holds when we look for a τ-transition, with one addition. If ρ can advance with a non-τ action, then we should look for a matching action in the other component that may have caused the two components to communicate.

To define the transformation for formulae of the form $\langle\leftarrow\alpha\rangle\varphi$, we again look at several cases separately. First we consider the case when ρ has the empty path. In this case it is obvious that no backward step is possible.

$$(\langle\leftarrow\alpha\rangle\varphi)/(p,\lambda) = \bot$$

The second case to consider is when ρ ends with a pseudo-transition, or a "gap". In this case the only possibility is that the other component (the one we are testing) is able to perform the backward transition.

$$(\langle\leftarrow\alpha\rangle\varphi)/\rho'(p' \dashrightarrow p') = \langle\leftarrow\alpha\rangle(\varphi/\rho')$$

The third case applies when ρ does indeed end with the transition we look for. In this case the other component must end with a matching gap.

$$(\langle\leftarrow\alpha\rangle\varphi)/\rho'(p'' \xrightarrow{\alpha} p') = \langle\dashleftarrow\rangle(\varphi/\rho') \tag{3.4}$$

The only remaining case to consider is when ρ ends with a transition different from the one we look for. We split this case further and consider again separately the cases when $\alpha \in A$ and when $\alpha = \tau$. The former case is simple: if ρ indicates that the last transition has a label other than the one specified in the diamond operator, there is no way that the composed computation satisfies it.

$$(\langle\leftarrow a\rangle\varphi)/\rho'(p'' \xrightarrow{\beta} p') = \bot \quad \text{where } a \neq \beta$$

If however the diamond operator mentions a τ transition, then we must look for a transition in the other component that can synchronise with the last one of ρ.

Note that this case does not include computations ending with a τ transition, as that case is covered by equation 3.4.

$$(\langle\leftarrow\tau\rangle\varphi)/\rho'(p'' \xrightarrow{b} p') = \langle\leftarrow\bar{b}\rangle(\varphi/\rho')$$

This covers all possible cases for $\langle\leftarrow\alpha\rangle\varphi/\rho$.

For the new operators, $\langle\dashrightarrow\rangle$ and $\langle\dashleftarrow\rangle$, the transformation is simple. First, if a composed computation should satisfy $\langle\dashrightarrow\rangle\varphi$, then it must be because both components are able to add a pseudo step and satisfy φ. I.e.

$$(\langle\dashrightarrow\rangle\varphi)/\rho = \langle\dashrightarrow\rangle\,(\varphi/\rho(p' \dashrightarrow p'))$$

where again $p' = \text{last}(\rho)$. If the composition should end with a $\dashrightarrow$ pseudo-transition, then both components must also end with $\dashrightarrow$. Thus, we have to consider two cases, where ρ does end with such a pseudo-transition and when it doesn't.

$$(\langle\dashleftarrow\rangle\varphi)/\rho = \begin{cases}\langle\dashleftarrow\rangle(\varphi/\rho') & \text{if } \rho = \rho'(p' \dashrightarrow p') \\ \bot & \text{otherwise}\end{cases}$$

The complete quotienting transformation is summarised in Table 3.1.

3.4.1 Decomposition theorem

Before we state and prove our main theorem, we establish a few useful lemmas.

Lemma 3.6. If $p \parallel q \xrightarrow{\alpha} p' \parallel q'$ where $p \not\equiv p'$ and $q \not\equiv q'$ then $\alpha = \tau$.

Proof. Consider the proof tree for the transition $p \parallel q \xrightarrow{\alpha} p' \parallel q'$ and, in particular, the last rule used in the proof. This rule can be one of the three rules for the parallel operator. The first two, where only one component advances, are ruled out since then either $p \equiv p'$ or $q \equiv q'$ must hold. Therefore the last rule used in the proof must be the communication rule, in which case the label of the proved transition can only be τ. □

Lemma 3.7. Let p, q be processes, $(p \parallel q, \pi) \in \mathcal{C}(p \parallel q)$ and $(\mu_1, \mu_2) \in D(\pi)$.
(i) If $(p \parallel q, \pi) \xrightarrow{\alpha} (p \parallel q, \pi')$ then there exists a pair $(\mu_1', \mu_2') \in D(\pi')$ such that one of the following holds.

1. $(p, \mu_1) \xrightarrow{\alpha} (p, \mu_1')$ and $(q, \mu_2) \dashrightarrow (q, \mu_2')$,
2. $(p, \mu_1) \dashrightarrow (p, \mu_1')$ and $(q, \mu_2) \xrightarrow{\alpha} (q, \mu_2')$ or
3. $\alpha = \tau$, $(p, \mu_1) \xrightarrow{a} (p, \mu_1')$ and $(q, \mu_2) \xrightarrow{\bar{a}} (q, \mu_2')$ for some $a \in A$.

(ii) Symmetrically,

1. If there exists a μ_1' s.t. $(p, \mu_1) \xrightarrow{\alpha} (p, \mu_1')$ then there exists a π' s.t. $(p \parallel q, \pi) \xrightarrow{\alpha} (p \parallel q, \pi')$ and $(\mu_1', \mu_2(q' \dashrightarrow q')) \in D(\pi')$ where $q' = \text{last}(\mu_2)$.

$$
\begin{aligned}
\top/\rho &= \top \\
(\varphi_1 \wedge \varphi_2)/\rho &= \varphi_1/\rho \wedge \varphi_2/\rho \\
(\neg\varphi)/\rho &= \neg(\varphi/\rho) \\
(\langle a\rangle\varphi)/\rho &= \langle a\rangle\,(\varphi/\rho(p' \dashrightarrow p')) \vee \left(\bigvee_{\rho':\rho \overset{a}{\to} \rho'} \langle\dashrightarrow\rangle(\varphi/\rho') \right) \\
(\langle \tau\rangle\varphi)/\rho &= \langle \tau\rangle\,(\varphi/\rho(p' \dashrightarrow p')) \vee \left(\bigvee_{\rho':\rho \overset{\tau}{\to} \rho'} \langle\dashrightarrow\rangle(\varphi/\rho') \right) \\
&\qquad \vee \left(\bigvee_{\rho',a:\rho \overset{a}{\to} \rho'} \langle\bar{a}\rangle(\varphi/\rho') \right) \\
(\langle \leftarrow \alpha\rangle\varphi)/(p,\lambda) &= \bot \\
(\langle \leftarrow \alpha\rangle\varphi)/\rho'(p' \dashrightarrow p') &= \langle \leftarrow \alpha\rangle(\varphi/\rho') \\
(\langle \leftarrow \alpha\rangle\varphi)/\rho'(p'' \xrightarrow{\alpha} p') &= \langle \dashleftarrow\rangle(\varphi/\rho') \\
(\langle \leftarrow a\rangle\varphi)/\rho'(p'' \xrightarrow{\beta} p') &= \bot \quad \text{where } a \neq \beta \\
(\langle \leftarrow \tau\rangle\varphi)/\rho'(p'' \xrightarrow{b} p') &= \langle \leftarrow \bar{b}\rangle(\varphi/\rho') \\
(\langle\dashrightarrow\rangle\varphi)/\rho &= \langle\dashrightarrow\rangle\,(\varphi/\rho(p' \dashrightarrow p')) \\
(\langle\dashleftarrow\rangle\varphi)/\rho &= \begin{cases} \langle\dashleftarrow\rangle(\varphi/\rho') & \text{if } \rho = \rho'(p' \dashrightarrow p') \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Table 3.1: Quotienting transformations of formulae in $HML^{*}_{\leftarrow}$

2. If there exists a μ_2' s.t. $(q,\mu_2) \xrightarrow{\alpha} (q,\mu_2')$ then there exists a π' s.t. $(p \parallel q,\pi) \xrightarrow{\alpha} (p \parallel q,\pi')$ and $(\mu_1(p' \dashrightarrow p'),\mu_2') \in D(\pi')$ where $p' = \text{last}(\mu_1)$.

3. If there exist μ_1' and μ_2' s.t. $(p,\mu_1) \xrightarrow{a} (p,\mu_1')$ and $(q,\mu_2) \xrightarrow{\bar{a}} (q,\mu_2')$ for some $a \in A$, then there exists π' s.t. $(p \parallel q,\pi) \xrightarrow{\tau} (p \parallel q,\pi')$ and $(\mu_1',\mu_2') \in D(\pi')$.

Proof. (i) Assume that $(p \parallel q,\pi) \xrightarrow{\alpha} (p \parallel q,\pi')$ and let $(\mu_1,\mu_2) \in D(\pi)$. This means there exist processes p',q',p'',q'' with $\pi' = \pi(p'' \parallel q'' \xrightarrow{\alpha} p' \parallel q')$, $p'' = \text{last}(\mu_1)$, $q'' = \text{last}(\mu_2)$. Since $p'' \parallel q'' \not\equiv p' \parallel q'$ we observe that $p'' \equiv p'$ and $q'' \equiv q'$ cannot hold simultaneously, so we consider the remaining cases.

1. $p'' \not\equiv p'$ and $q'' \equiv q'$. In this case the transition $p'' \parallel q' \xrightarrow{\alpha} p' \parallel q'$ was proven using the first rule for $\parallel$. Its only premise must hold, namely

$p'' \xrightarrow{\alpha} p'$. We therefore let $\mu_1' = \mu_1(p'' \xrightarrow{\alpha} p')$ and $\mu_2' = \mu_2(q' \dashrightarrow q')$. From the inductive definition of D it is easy to see that $(\mu_1', \mu_2') \in D(\pi')$.

2. $p'' \equiv p'$ and $q'' \not\equiv q'$. This case is entirely symmetric to the previous one where the proof is based on the second rule for $\|$.

3. $p'' \not\equiv p'$ and $q'' \not\equiv q'$. Here the proof of the transition $p'' \parallel q'' \xrightarrow{\alpha} p' \parallel q'$ must be based on the third rule for $\|$, namely the communication rule and $\alpha = \tau$, as seen by Lemma 3.6. By the premises of this rule there exists an $a \in A$ such that $p'' \xrightarrow{a} p'$ and $q'' \xrightarrow{\bar{a}} q'$. We simply let $\mu_1' = \mu_1(p'' \xrightarrow{a} p')$ and $\mu_2' = \mu_2(q'' \xrightarrow{\bar{a}} q')$. Again it is clear from the definition of D that $(\mu_1', \mu_2') \in D(\pi')$.

(ii) The construction of π' in all cases is straightforward and unique (cfr. Lemma 3.4). The rest is simple to check with the definition of D. □

Lemma 3.8. Let $(p \parallel q, \pi) \in \mathcal{C}(p \parallel q)$ with π non-empty and $(\mu_1, \mu_2) \in D(\pi)$. Let π', μ_1' and μ_2' be the prefixes of length $|\pi| - 1$ of π, μ_1 and μ_2 respectively. Then $(\mu_1', \mu_2') \in D(\pi')$.

Proof. Follows directly from the definition of D. □

We are now ready to prove the main theorem in this chapter, to the effect that the quotienting of a formula φ with respect to a computation ρ is properly defined.

Theorem 3.9. For CCS processes p, q and a computation $(p \parallel q, \pi) \in \mathcal{C}(p \parallel q)$ and a formula $\varphi \in HML^*_{\leftarrow}$ we have

$$(p \parallel q, \pi) \vDash^* \varphi \quad \Rightarrow \quad \forall(\mu_1, \mu_2) \in D(\pi) : (p, \mu_1) \vDash^* \varphi/(q, \mu_2) \tag{3.5}$$

and conversely (note the direction of the implication),

$$(p \parallel q, \pi) \vDash^* \varphi \quad \Leftarrow \quad \exists(\mu_1, \mu_2) \in D(\pi) : (p, \mu_1) \vDash^* \varphi/(q, \mu_2). \tag{3.6}$$

Proof. We prove both implications simultaneously by induction on the structure of φ. In the following text, the terms *"the left-hand side"* and *"the right-hand side"* refer respectively to the left- and right-hand sides of the above implications where the quantifier used in the right-hand side will be made clear by the context.

Case $\varphi = \top$ Then $\varphi/(q, \mu_2) = \top$ and both sides of both (3.5) and (3.6) are trivially satisfied.

Case $\varphi = \psi_1 \wedge \psi_2$

($\Rightarrow$) First assume $(p \parallel q, \pi) \vDash^* \psi_1 \wedge \psi_2$ and let $(\mu_1, \mu_2) \in D(\pi)$. Since both ψ_1 and ψ_2 are smaller than φ and both are satisfied by $(p \parallel q, \pi)$ we have by induction that $(p, \mu_1) \vDash^* \psi_i/(q, \mu_2)$ for $i \in \{1, 2\}$. Since $\varphi/(q, \mu_2) = (\psi_1 \wedge \psi_2)/(q, \mu_2) = \psi_1/(q, \mu_2) \wedge \psi_2/(q, \mu_2)$ we obtain $(p, \mu_1) \vDash^* \varphi/(q, \mu_2)$.

($\Leftarrow$) Now assume the right side of (3.6),

$$\exists(\mu_1, \mu_2) \in D(\pi) : (p, \mu_1) \vDash^* (\psi_1 \wedge \psi_2)/(q, \mu_2).$$

By definition the formula is equal to $\psi_1/(q,\mu_1) \wedge \psi_2/(q,\mu_2)$. By induction $(p \parallel q, \pi)$ satisfies both ψ_1 and ψ_2 and thus also $\psi_1 \wedge \psi_2 = \varphi$.

Case $\varphi = \neg\psi$

$(\Rightarrow)$ First assume the left side $(p \parallel q, \pi) \vDash^* \neg\psi$. Assume towards contradiction that there does exist a decomposition $(\mu_1', \mu_2') \in D(\pi)$ such that $(p, \mu_1') \vDash^* \psi/(q, \mu_2')$. Then by induction (3.6) gives $(p \parallel q, \pi) \vDash^* \psi$, which is in direct contradiction with our assumption. Since no such decomposition can exist, it holds for all $(\mu_1, \mu_2) \in D(\pi)$ that $(p, \mu_1) \vDash^* \neg\psi/(q, \mu_2) = \varphi/(q, \mu_2)$.

$(\Leftarrow)$ Assume the right side of (3.6), namely there exists a decomposition $(\mu_1, \mu_2) \in D(\pi)$ such that $(p, \mu_1) \vDash^* \neg\psi/(q, \mu_2)$. Assume, again towards a contradiction, that $(p \parallel q, \pi) \vDash^* \psi$. By induction, (3.5) then gives that for all $(\mu_1', \mu_2') \in D(\pi)$, $(p, \mu_1') \vDash^* \psi/(q, \mu_2')$. In particular, this holds for the decomposition (μ_1, μ_2), which contradicts our assumption. Therefore we must have that $(p \parallel q, \pi) \vDash^* \neg\psi = \varphi$.

Case $\varphi = \langle\alpha\rangle\psi$

$(\Rightarrow)$ Again, first assume the left side and take $(\mu_1, \mu_2) \in D(\pi)$. Then there exists a computation $(p \parallel q, \pi')$ s.t. $(p \parallel q, \pi) \xrightarrow{\alpha} (p \parallel q, \pi')$ and $(p \parallel q, \pi') \vDash^* \psi$. By part (i) of Lemma 3.7 there exists a pair $(\mu_1', \mu_2') \in D(\pi')$. Since ψ is a subformula of φ we have by induction that

$$(p, \mu_1') \vDash^* \psi/(q, \mu_2') \tag{3.7}$$

Lemma 3.7 also states that one of the following three cases holds.

1. $(p, \mu_1) \xrightarrow{\alpha} (p, \mu_1')$ and $(q, \mu_2) \dashrightarrow (q, \mu_2')$. From (3.7) we have that $(p, \mu_1) \vDash^* \langle\alpha\rangle(\psi/(q, \mu_2'))$ and since the formula $\langle\alpha\rangle(\psi/(q, \mu_2'))$ is the first clause of the disjunction defining $\varphi/(q, \mu_2)$ then also $(p, \mu_1) \vDash^* \varphi/(q, \mu_2)$.

2. $(p, \mu_1) \dashrightarrow (p, \mu_1')$ and $(q, \mu_2) \xrightarrow{\alpha} (q, \mu_2')$. Again from (3.7) we have that $(p, \mu_1) \vDash^* \langle\dashrightarrow\rangle(\psi/(q, \mu_2'))$, and again the formula $\langle\dashrightarrow\rangle(\psi/(q, \mu_2'))$ is a clause of the disjunction defining $\varphi/(q, \mu_2)$ so $(p, \mu_1) \vDash^* \varphi/(q, \mu_2)$.

3. $\alpha = \tau$, $(p, \mu_1) \xrightarrow{a} (p, \mu_1')$ and $(q, \mu_2) \xrightarrow{\bar{a}} (q, \mu_2')$ for some $a \in A$. Then the disjunction $\varphi/(q, \mu_2)$ has a clause $\langle a\rangle\,(\psi/(q, \mu_2'))$ (note that $\bar{\bar{a}} = a$). By (3.7) we get that $(p, \mu_1) \vDash^* \varphi/(q, \mu_2)$.

In all cases the result is the same, namely $(p, \mu_1) \vDash^* \varphi/(q, \mu_2)$ which is what we wanted to prove.

$(\Leftarrow)$ Now assume the right side of (3.6), i.e. there exists a $(\mu_1, \mu_2) \in D(\pi)$ such that that $(p, \mu_1) \vDash^* \langle\alpha\rangle\psi/(q, \mu_2)$. We know $\langle\alpha\rangle\psi/(q, \mu_2)$ is a disjunction of one or more clauses so (p, μ_1) must satisfy at least one of them. Each clause has one of three forms, and we analyze the possible cases. Let φ' be a clause that (p, μ_1) satisfies.

1. Assume that $\varphi' = \langle\alpha\rangle\,(\psi/(q, \mu_2)(q' \dashrightarrow q'))$ where $q' = \text{last}(q, \mu_2)$. Then there is a μ_1' such that $(p, \mu_1) \xrightarrow{\alpha} (p, \mu_1')$ and $(p, \mu_1') \vDash^* \psi/(q, \mu_2(q' \dashrightarrow q'))$.

If we let $\mu_2' = \mu_2(q' \dashrightarrow q')$ then part (ii) of Lemma 3.7 gives that there exists a π' with $(\mu_1', \mu_2') \in D(\pi')$ and $(p \parallel q, \pi) \xrightarrow{\alpha} (p \parallel q, \pi')$. Since $(p, \mu_1') \models^* \psi/(q, \mu_2')$ then by induction, since ψ is smaller than φ, $(p \parallel q, \pi') \models^* \psi$. This in turn means that $(p \parallel q, \pi) \models^* \langle\alpha\rangle\psi = \varphi$.

2. Assume that $\varphi' = \langle \dashrightarrow \rangle (\psi/(q, \mu_2'))$ for some μ_2' such that $(q, \mu_2) \xrightarrow{\alpha} (q, \mu_2')$. Let $\mu_1' = \mu_1(p' \dashrightarrow p')$ where $p' = \mathrm{last}(p, \mu_1)$. Lemma 3.7 gives the existence of π' with $(\mu_1', \mu_2') \in D(\pi')$ and $(p \parallel q, \pi) \xrightarrow{\alpha} (p \parallel q, \pi')$. Since $(p, \mu_1') \models^* \psi/(q, \mu_2')$ then by induction $(p \parallel q, \pi') \models^* \psi$ and thus $(p \parallel q, \pi) \models^* \langle\alpha\rangle\psi = \varphi$.

3. Assume that $\alpha = \tau$ and $\varphi' = \langle \bar{a} \rangle (\psi/(q, \mu_2'))$ for some μ_2' s.t. $(q, \mu_2) \xrightarrow{a} (q, \mu_2')$ and $a \in A$. This means there is a μ_1' with $(p, \mu_1) \xrightarrow{\bar{a}} (p, \mu_1')$. Lemma 3.7 then says that there exists π' with $(p \parallel q, \pi) \xrightarrow{\tau} (p \parallel q, \pi')$ and $(\mu_1', \mu_2') \in D(\pi')$. Since $(p, \mu_1') \models^* \psi/(q, \mu_2')$ then by induction $(p \parallel q, \pi') \models^* \psi$ and thus $(p \parallel q, \pi) \models^* \langle\tau\rangle\psi = \varphi$.

In all cases we obtain what we wanted to prove, namely $(p \parallel q, \pi) \models^* \varphi$.

Case $\varphi = \langle\leftarrow\alpha\rangle\psi$

$(\Rightarrow)$ Assume that $(p \parallel q, \pi) \models^* \langle\leftarrow\alpha\rangle\psi$ and take $(\mu_1, \mu_2) \in D(\pi)$. Since $(p \parallel q, \pi') \xrightarrow{\alpha} (p \parallel q, \pi)$ for some π' such that $(p \parallel q, \pi') \models^* \psi$, there exist processes p', q', p'', q'' such that $\pi = \pi'(p'' \parallel q'' \xrightarrow{\alpha} p' \parallel q')$. By analysing the definition of D, we can gain some information about μ_1 and μ_2, in particular by comparing p'' to p' and q'' to q'. Since $p'' \parallel q'' \not\equiv p' \parallel q'$ we must consider three cases.

1. $p'' \not\equiv p'$ and $q'' \equiv q'$. Then $(\mu_1, \mu_2) = (\mu_1'(p'' \xrightarrow{\alpha} p'), \mu_2'(q' \dashrightarrow q'))$ for some $(\mu_1', \mu_2') \in D(\pi')$. Given this form of μ_2 we also know that $((\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2) = \langle\leftarrow\alpha\rangle (\psi/(q, \mu_2'))$. Since $(p \parallel q, \pi') \models^* \psi$, we get by induction that $(p, \mu_1') \models^* \psi/(q, \mu_2')$, which in turn means that $(p, \mu_1) \models^* \langle\leftarrow\alpha\rangle (\psi/(q, \mu_2))$ and since the last step of μ_2 is a pseudo-step, $\langle\leftarrow\alpha\rangle (\psi/(q, \mu_2)) = ((\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2)$.

2. $p'' \equiv p'$ and $q'' \not\equiv q'$. In this case $(\mu_1, \mu_2) = (\mu_1'(p' \dashrightarrow p'), \mu_2'(q'' \xrightarrow{\alpha} q'))$ where $(\mu_1', \mu_2') \in D(\pi')$. This form of μ_2 means that $\langle\leftarrow\alpha\rangle\psi/(q, \mu_2) = \langle \leftarrow\!\!- \rangle (\psi/(q, \mu_2'))$. By induction, the fact that $(p \parallel q, \pi') \models^* \psi$ gives that $(p, \mu_1') \models^* \psi/(q, \mu_2')$, so since $\mu_2 = \mu_2'(q'' \xrightarrow{\alpha} q')$ holds, then we have that $(p, \mu_1) \models^* \langle \leftarrow\!\!- \rangle (\psi/(q, \mu_2')) = ((\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2)$.

3. $p'' \not\equiv p'$ and $q'' \not\equiv q'$. By Lemma 3.6 α must be equal to τ. Thus we have that $(\mu_1, \mu_2) = (\mu_1'(p'' \xrightarrow{a} p'), \mu_2'(q'' \xrightarrow{\bar{a}} q'))$ for some $a \in A$ and $(\mu_1', \mu_2') \in D(\pi')$. This also means that $((\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2) = \langle\leftarrow a\rangle (\psi/(q, \mu_2'))$. Since $(p \parallel q, \pi') \models^* \psi$ we again have by induction that $(p, \mu_1') \models^* \psi/(q, \mu_2')$. We therefore obtain that $(p, \mu_1) \models^* \langle\leftarrow a\rangle (\psi/(q, \mu_2')) = ((\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2)$.

In all cases we obtain the same result, namely $(p, \mu_1) \models^* ((\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2) = \varphi/(q, \mu_2)$.

($\Leftarrow$) Now assume that there is $(\mu_1, \mu_2) \in D(\pi)$ s.t. $(p, \mu_1) \vDash^* (\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2)$. This means that μ_1 and μ_2 are non-empty. By comparing α with the last transition of μ_2 we can infer the form of $(\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2)$.

- If the last transition of μ_2 is a $\dashrightarrow$ transition, i.e. $\mu_2 = \mu_2'(q' \dashrightarrow q')$ for some μ_2 and $q' = \text{last}(q, \mu_2)$, then we know that $(\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2) = \langle\leftarrow\alpha\rangle\,(\psi/(q, \mu_2'))$. By our assumption this is satisfied by (p, μ_1) so there exists a μ_1' s.t. $(p, \mu_1') \xrightarrow{\alpha} (p, \mu_1)$ and $(p, \mu_1') \vDash^* \psi/(q, \mu_2')$. Let π' be π without the last transition (note that π is non-empty since μ_1 and μ_2 are). By Lemma 3.8 $(\mu_1', \mu_2') \in D(\pi')$ and by induction we have that $(p \parallel q, \pi') \vDash^* \psi$. From the definition of D we can also see that the last transition of π can only be $\xrightarrow{\alpha}$. Thus $(p \parallel q, \pi') \xrightarrow{\alpha} (p \parallel q, \pi)$ so $(p \parallel q, \pi) \vDash^* \langle\leftarrow\alpha\rangle\psi$.

- If the last transition of μ_2 is an $\xrightarrow{\alpha}$ transition, i.e. one having the same label as the formula is testing for, then $(\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2) = \langle\leftarrow\dashrightarrow\rangle\,(\psi/(q, \mu_2'))$ where μ_2' is μ_2 without the last transition. Note that $(q, \mu_2') \xrightarrow{\alpha} (q, \mu_2)$. Since (p, μ_1) satisfies this formula there is a μ_1' s.t. $(p, \mu_1') \dashrightarrow (p, \mu_1)$ and $(p, \mu_1) \vDash^* \psi/(q, \mu_2')$. By Lemma 3.8 $(\mu_1', \mu_2') \in D(\pi')$ where π' is again π without the last transition. Also again, we can see from the definition of D that $(p \parallel q, \pi') \xrightarrow{\alpha} (p \parallel q, \pi)$. By induction it thus holds that $(p \parallel q, \pi') \vDash^* \psi$ and so $(p \parallel q, \pi) \vDash^* \langle\leftarrow\alpha\rangle\psi$.

- The only remaining case to consider is when μ_2 ends with a transition $\xrightarrow{\beta}$ where $\beta \neq \alpha$. Then α can only be τ, since otherwise the formula $(\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2)$ equals $\bot$, which contradicts our assumption that (p, μ_1) satisfies it. Since $\beta \neq \alpha = \tau$ we also know β must be some label $a \in A$. This means that $(\langle\leftarrow\alpha\rangle\psi)/(q, \mu_2) = \langle\leftarrow\bar{a}\rangle\,(\psi/(q, \mu_2'))$ where μ_2' is yet again μ_2 without the last transition. Since (p, μ_1) satisfies this formula, there is a μ_1' s.t. $(p, \mu_1') \xrightarrow{\bar{a}} (p, \mu_1)$ and $(p, \mu_1') \vDash^* \psi/(q, \mu_2')$. By Lemma 3.8, $(\mu_1', \mu_2') \in D(\pi')$ where π' is π without the last transition. By induction, $(p \parallel q, \pi') \vDash^* \psi$ and from the definition of D we can see that $(p \parallel q, \pi') \xrightarrow{\tau} (p \parallel q, \pi)$ is the only possible transition between the two computations. Therefore, $(p \parallel q, \pi) \vDash^* \langle\leftarrow\tau\rangle\psi = \langle\leftarrow\alpha\rangle\psi$.

In all cases $(p \parallel q, \pi) \vDash^* \langle\leftarrow\alpha\rangle\psi = \varphi$.

Case $\varphi = \langle\dashrightarrow\rangle\psi$

($\Rightarrow$) First assume $(p \parallel q, \pi) \vDash^* \langle\dashrightarrow\rangle\psi$ and take $(\mu_1, \mu_2) \in D(\pi)$. This means there exists a π' s.t. $(p \parallel q, \pi) \dashrightarrow (p \parallel q, \pi')$ and $(p \parallel q, \pi') \vDash^* \psi$. By definition $(\langle\dashrightarrow\rangle\psi)/(q, \mu_2) = \langle\dashrightarrow\rangle\,(\psi/(q, \mu_2'))$, where we let $(\mu_1', \mu_2') = (\mu_1(p' \dashrightarrow p'), \mu_2(q' \dashrightarrow q'))$ with $(p' \parallel q') = \text{last}(p \parallel q, \pi)$. This is according to the definition of D so $(\mu_1', \mu_2') \in D(\pi')$. Thus, by induction $(p, \mu_1') \vDash^* \psi/(q, \mu_2')$. Since $(p, \mu_1) \dashrightarrow (p, \mu_1')$ we obtain that $(p, \mu_1) \vDash^* \langle\dashrightarrow\rangle\,(\psi/(q, \mu_2')) = (\langle\dashrightarrow\rangle\psi)/(q, \mu_2)$.

($\Leftarrow$) Assume that $\exists(\mu_1, \mu_2) \in D(\pi) : (p, \mu_1) \vDash^* (\langle\dashrightarrow\rangle\psi)/(q, \mu_2)$. We want to show that $(p \parallel q, \pi) \vDash^* \langle\dashrightarrow\rangle\psi$. Let $p' = \text{last}(\mu_1)$ and $q' = \text{last}(\mu_2)$. The formula $\langle\dashrightarrow\rangle\psi)/(q, \mu_2)$ is equal to $\langle\dashrightarrow\rangle\,(\psi/(q, \mu_2'))$ where $\mu_2' = \mu_2(q' \dashrightarrow q')$. If we let $\pi' = \pi(p' \parallel q' \dashrightarrow p' \parallel q')$ and $\mu_1' = \mu_1(p' \dashrightarrow p')$, then, by definition of D, $(\mu_1', \mu_2') \in D(\pi')$. Observe that $(p, \mu_1) \dashrightarrow (p, \mu_1')$ and that the $\dashrightarrow$ relation is deterministic. Therefore $(p, \mu_1') \vDash^* \psi/(q, \mu_2')$ for each (μ_1, μ_2). Induction gives that $(p \parallel q, \pi') \vDash^* \psi$. Now it follows trivially that $(p \parallel q, \pi) \vDash^* \langle \dashrightarrow \rangle \psi$ because $(p \parallel q, \pi) \dashrightarrow (p \parallel q, \pi')$.

Case $\varphi = \langle \dashleftarrow \rangle \varphi$

($\Rightarrow$) Assume $(p \parallel q, \pi) \vDash^* \langle \dashleftarrow \rangle \psi$ and take $(\mu_1, \mu_2) \in D(\pi)$. This means that $\pi = \pi'(p' \parallel q' \dashrightarrow p' \parallel q')$ and $(p \parallel q, \pi') \vDash^* \psi$, where $p' \parallel q' = \text{last}(\pi)$. It is obvious, from the definition of D that μ_1 and μ_2 both end with $\dashrightarrow$ since π ends with $\dashrightarrow$. Let π', μ_1', μ_2' be π, μ_1, μ_2 without their last transition respectively (note that our assumption guarantees that they are non-empty). By Lemma 3.8 we know that $(\mu_1', \mu_2') \in D(\pi')$. Since $(p \parallel q, \pi') \vDash^* \psi$, we have by induction that $(p, \mu_1') \vDash^* \psi/(q, \mu_2')$. Then $(p, \mu_1) \vDash^* \langle \dashleftarrow \rangle (\psi/(q, \mu_2')) = (\langle \dashleftarrow \rangle \psi)/(q, \mu_2)$.

($\Leftarrow$) Now assume $\exists(\mu_1, \mu_2) \in D(\pi) : (p, \mu_1) \vDash^* (\langle \dashleftarrow \rangle \psi)/(q, \mu_2)$. Then the last step of μ_2 is $\dashrightarrow$ since otherwise the formula would be equal to $\bot$, which could not be satisfied by (p, μ_1). We see furthermore that the quotiented formula is $\langle \dashleftarrow \rangle (\psi/(q, \mu_2'))$ where μ_2' is again μ_2 without its last step. This means the last step of μ_1 is also $\langle \dashrightarrow \rangle$ (a fact we could also have deduced from the definition of D). Let μ_1' be μ_1 without this step. If we also let π' be π without the last step, then by Lemma 3.8 we have $(\mu_1', \mu_2') \in D(\pi')$. Since $(p, \mu_1') \vDash^* \psi/(q, \mu_2')$ induction gives that $(p \parallel q, \pi') \vDash^* \psi$. By the definition of D we see that the last step of π can only be $\dashrightarrow$ so $(p \parallel q, \pi) \vDash^* \langle \dashleftarrow \rangle \psi$.

This concludes the analysis of all structural forms for φ. In each case we have shown by structural induction that each direction of the theorem holds. □

Theorem 3.9 uses the existential quantifier in the right-to-left direction. This makes it easy to show that a composed computation satisfies a formula, given only one witness of a decomposition with one component satisfying a rewritten formula. Note however that the set of decompositions of any given process is never empty, i.e. every parallel computation has a decomposition. This allows us to write the above theorem in a more symmetrical form.

Corollary 3.10. For CCS processes p, q, a parallel computation $(p \parallel q, \pi)$ and a formula $\varphi \in HML^*_{\leftarrow}$ we have

$$(p \parallel q, \pi) \vDash^* \varphi \quad \Leftrightarrow \quad \forall(\mu_1, \mu_2) \in D(\pi) : (p, \mu_1) \vDash^* \varphi/(q, \mu_2) \tag{3.8}$$

Proof. ($\Rightarrow$) This case follows directly from the theorem.

($\Leftarrow$) Assume that $\forall(\mu_1, \mu_2) \in D(\pi) : (p, \mu_1) \vDash^* \varphi/(q, \mu_2)$. Specifically, since there exists at least one decomposition $(\mu_1', \mu_2') \in D(\pi)$, the above holds for that particular decomposition. By the $\Leftarrow$ part of Theorem 3.9, we thus have that $(p \parallel q, \pi) \vDash^* \varphi$. □

3.5 Extensions and related work

So far we have stated and proven a decompositional theorem that allows us to apply decompositional reasoning for history-based computations over CCS and the logic $HML_{\leftarrow}$. However, more work remains to be done in order to apply this theory to meaningful examples. Part of this work is well underway already although the details have not been worked out fully for this particular thesis submission. In particular, we have extended the decompositional theorem to a recursive logic $HML^{*}_{\leftarrow,\mathcal{X}}$ (equivalent to extending the modal μ-calculus with past), which allows a much wider class of interesting properties to be specified as fixed-points of systems of recursive logic equations.

In the decomposition of computations, we rely on some specific properties of CCS at the syntactic level, namely to detect which rule of the parallel operator was applied. By tagging transitions with their proofs [Boudol and Castellani 1988, Degano and Priami 1992], or even just the last proof used, we could eliminate this restriction and extend our approach to more general languages involving parallel composition. Another possibility is to construct a rule format that guarantees the properties we use at a more general level, inspired by the work of [Fokkink et al. 2006].

In this work we have only considered parallel composition. However, decompositional results have been shown for the more general setting of *process contexts* [Larsen and Xinxin 1991]. The two parts considered as components are then not confined to components of the parallel construction, but a context $C[\cdot]$ (a process term with a *hole*) and general process p to instantiate the context with. A general property of the instantiated context $C[p]$ can the be transformed into an equivalent property of p, where the transformation depends on C. As the state space explosion of model-checking problems is often due to use of the parallel construct, we deemed our approach as a useful step towards a full decompositional result, since the decomposition of computations will be more complex for general contexts.

The initial motivation for this work was the application of epistemic logic to behavioural models [Dechesne et al. 2007]. We would therefore like to extend our results to logics that include epistemic operators, reasoning about the knowledge of agents observing a running system. This work depends somewhat on our extensions to recursive formulae.

Our work is based on many previous results, both in decompositional reasoning as well as history-based process systems. Yet, to the best of our knowledge, we are the first to combine the two.

Our notion of computations comes from [Nicola et al. 1990], where they are used to compare bisimulations, that consider both forwards and backward actions, to more conventional forward-only bisimulations. The backwards modality of $HML_{\leftarrow}$comes from [Hennessy and Stirling 1985], which shares the same notion of computations as well. Our methods for decompositional reasoning largely build on and adapt the methods from [Ingólfsdóttir et al. 1987, Larsen and Xinxin 1991, Fokkink et al. 2003]. However, as the work presented in this chapter shows, the development of a theory of decompositional reasoning in a setting with past modalities involves subtleties and design decisions that do not arise in previous work on HML and Kozen's μ-calculus [Kozen 1983].

chapter four

Rule Formats for Determinism and Idempotency

joint work with Luca Aceto, Anna Ingólfsdottir, MohammadReza Mousavi and Michel Reniers

4.1 Introduction

Structural Operational Semantics (SOS) [Plotkin 2004a] is a popular method for assigning a rigorous meaning to specification and programming languages. The meta-theory of SOS provides powerful tools for proving semantic properties for such languages without investing too much time on the actual proofs; it offers syntactic templates for SOS rules, called *rule formats*, which guarantee semantic properties once the SOS rules conform to the templates (see, e.g., the references [Aceto et al. 2001, Mousavi et al. 2007] for surveys on the meta-theory of SOS). There are various rule formats in the literature for many different semantic properties, ranging from basic properties such as commutativity [Mousavi et al. 2005] and associativity [Cranen et al. 2008] of operators, and congruence of behavioral equivalences (see, e.g., [Verhoef 1995]) to more technical and involved ones such as non-interference [Tini 2004] and (semi-)stochasticity [Lanotte and Tini 2005]. In this chapter, we propose rule formats for two (related) properties, namely, determinism and idempotency.

Determinism is a semantic property of (a fragment of) a language that specifies that a program cannot evolve operationally in several different ways. It holds for sub-languages of many process calculi and programming languages, and it is also a crucial property for many formalisms for the description of timed systems, where time transitions are required to be deterministic, because the passage of time should not resolve any choice.

Idempotency is a property of binary composition operators requiring that the composition of two identical specifications or programs will result in a piece of specification or program that is equivalent to the original components. Idempotency of a binary operator f is concisely expressed by the following algebraic equation.

$$f(x,x) = x$$

Determinism and idempotency may seem unrelated at first sight. However, it turns out that in order to obtain a powerful rule format for idempotency, we need to have the determinism of certain transition relations in place. Therefore, having a syntactic condition for determinism, apart from its intrinsic value, results in a powerful, yet syntactic framework for idempotency.

To our knowledge, our rule format for idempotency has no precursor in the literature. As for determinism, in [Fokkink and Vu 2003], a rule format for bounded nondeterminism is presented but the case for determinism is not studied. Also, in [Ulidowski and Yuen 1997] a rule format is proposed to guarantee several time-related properties, including time determinism, in the settings of Ordered SOS. In case of time determinism, their format corresponds to a subset of our rule format when translated to the setting of ordinary SOS, by means of the recipe given in [Mousavi et al. 2006].

We made a survey of existing deterministic process calculi and of idempotent binary operators in the literature and we have applied our formats to them. Our formats could cover all practical cases that we have discovered so far, which is an indication of its expressiveness and relevance.

The rest of this chapter is organized as follows. In Section 4.2 we recall some basic definitions from the meta-theory of SOS. In Section 4.3, we present our rule format for determinism and prove that it does guarantee determinism for certain transition relations. Section 4.4 introduces a rule format for idempotency and proves it correct. In Sections 4.3 and 4.4, we also provide several examples to motivate the constraints of our rule formats and to demonstrate their practical applications. Finally, Section 4.5 concludes the chapter and presents some directions for future research.

4.2 Preliminaries

In this section we present, for sake of completeness, some standard definitions from the meta-theory of SOS that will be used in the remainder of the chapter.

Definition 4.1. *(Signature and terms)* We let V represent an infinite set of variables and use $x, x', x_i, y, y', y_i, \ldots$ to range over elements of V. A *signature* Σ is a set of function symbols, each with a fixed arity. We call these symbols *operators* and usually represent them by $f, g, \ldots$. An operator with arity zero is called a *constant*. We define the set $\mathbb{T}(\Sigma)$ of *terms* over Σ as the smallest set satisfying the following constraints.

- A variable $x \in V$ is a term.
- If $f \in \Sigma$ has arity n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

We use $t, t', t_i, \ldots$ to range over terms. We write $t_1 \equiv t_2$ if t_1 and t_2 are syntactically equal. The function $vars : \mathbb{T}(\Sigma) \to 2^V$ gives the set of variables appearing in a term. The set $\mathbb{C}(\Sigma) \subseteq \mathbb{T}(\Sigma)$ is the set of *closed terms*, i.e., terms that contain no variables. We use $p, p', p_i, \ldots$ to range over closed terms. A *substitution* σ is a function of type $V \to \mathbb{T}(\Sigma)$. We extend the domain of substitutions to terms homomorphically. If the range of a substitution lies in $\mathbb{C}(\Sigma)$, we say that it is a *closing substitution*.

Definition 4.2. *(Transition System Specifications (TSS), formulae and transition relations)* A *transition system specification* is a triplet (Σ, L, D) where

- Σ is a signature.
- L is a set of labels. If $l \in L$, and $t, t' \in \mathbb{T}(\Sigma)$ we say that $t \xrightarrow{l} t'$ is a *positive formula* and $t \overset{l}{\nrightarrow}$ is a *negative formula*. A formula, typically denoted by φ, ψ, φ', φ_i, … is either a negative formula or a positive one.
- D is a set of *deduction rules*, i.e., tuples of the form (Φ, φ) where Φ is a set of formulae and φ is a positive formula. We call the formulae contained in Φ the *premises* of the rule and φ the *conclusion*.

We write $vars(\varphi)$ and $vars(r)$ to denote the set of variables appearing in a formula φ and a deduction rule (r), respectively. We say a formula is *closed* if all of its terms are closed. Substitutions are also extended to formulae and sets of formulae in the natural way. A set of positive closed formulae is called a *transition relation*.

We often refer to a formula $t \xrightarrow{l} t'$ as a *transition* with t being its *source*, l its label, and t' its *target*. A deduction rule (Φ, φ) is typically written as $\frac{\Phi}{\varphi}$. For a deduction rule r, we write $conc(r)$ to denote its conclusion and $prem(r)$ to denote its premises. We call a deduction rule f*-defining* when the outermost function symbol appearing in its source of the conclusion is f.

The meaning of a TSS is defined by the following notion of least three-valued stable model. To define this notion, we need two auxiliary definitions, namely provable transition rules and contradiction, which are given below.

Definition 4.3. *(Provable Transition Rules)* A deduction rule is called a *transition rule* when it is of the form $\frac{N}{\varphi}$ with N a set of *negative formulae*. A TSS $\mathcal{T}$ *proves* $\frac{N}{\varphi}$, denoted by $\mathcal{T} \vdash \frac{N}{\varphi}$, when there is a well-founded upwardly branching tree with formulae as nodes and of which

- the root is labelled by φ;
- if a node is labelled by ψ and the nodes above it form the set K then:
  - ψ is a negative formula and $\psi \in N$, or
  - ψ is a positive formula and $\frac{K}{\psi}$ is an instance of a deduction rule in $\mathcal{T}$.

Definition 4.4. *(Contradiction and Contingency)* Formula $t \xrightarrow{l} t'$ is said to *contradict* $t \overset{l}{\nrightarrow}$, and vice versa. For two sets Φ and Ψ of formulae, Φ *contradicts* Ψ, denoted by $\Phi \nvDash \Psi$, when there is a $\varphi \in \Phi$ that contradicts a $\psi \in \Psi$. Φ is *contingent* w.r.t. Ψ, denoted by $\Phi \vDash \Psi$, when Φ does not contradict Ψ.

It immediately follows from the above definition that contradiction and contingency are symmetric relations on (sets of) formulae. We now have all the necessary ingredients to define the semantics of TSSs in terms of three-valued stable models.

Definition 4.5. *(The Least Three-Valued Stable Model)* A pair (C, U) of sets of positive closed transition formulae is called a *three-valued stable model* for a TSS $\mathcal{T}$ when

- for all $\varphi \in C$, $\mathcal{T} \vdash \frac{N}{\varphi}$ for a set N such that $C \cup U \vDash N$, and

- for all $\varphi \in U$, $\mathcal{T} \vdash \frac{N}{\varphi}$ for a set N such that $C \vDash N$.

C stands for *Certainly* and U for *Unknown*; the third value is determined by the formulae not in $C \cup U$. The *least* three-valued stable model is a three valued stable model which is the least with respect to the ordering on pairs of sets of formulae defined as $(C, U) \leq (C', U')$ iff $C \subseteq C'$ and $U' \subseteq U$. When for the least three-valued stable model it holds that $U = \{\}$, we say that $\mathcal{T}$ is *complete*.

Complete TSSs univocally define a transition relation, i.e., the C component of their least three-valued stable model. Completeness is central to almost all meta-results in the SOS meta-theory and, as it turns out, it also plays an essential role in our meta-results concerning determinism and idempotency. All practical instances of TSSs are complete and there are syntactic sufficient conditions guaranteeing completeness, see for example [Groote 1993].

4.3 Determinism

Definition 4.6. *(Determinism)* A transition relation T is called deterministic for label l, when if $p \xrightarrow{l} p' \in T$ and $p \xrightarrow{l} p'' \in T$, then $p' \equiv p''$.

Before we define a format for determinism, we need two auxiliary definitions. The first one is the definition of source dependent variables, which we borrow from [Mousavi and Reniers 2005] with minor additions.

Definition 4.7. *(Source dependency)* For a deduction rule, we define the set of *source dependent* variables as the smallest set that contains

1. all variables appearing in the source of the conclusion, and
2. all variables that appear in the target of a premise where all variables in the source of that premise are source dependent.

For a source dependent variable v, let $\mathcal{R}$ be the collection of transition relations appearing in a set of premises needed to show source dependency through condition 2. We say that v is source dependent *via* the relations in $\mathcal{R}$.

Note that for a source dependent variable, the set $\mathcal{R}$ is not necessarily unique. For example, in the rule

$$\frac{y \xrightarrow{l_1} y' \quad x \xrightarrow{l_2} z \quad z \xrightarrow{l_3} y'}{f(x, y) \xrightarrow{l} y'}$$

the variable y' is source dependent both via the set $\{\xrightarrow{l_1}\}$ as well as $\{\xrightarrow{l_2}, \xrightarrow{l_3}\}$.

The second auxiliary definition needed for our determinism format is the definition of determinism-respecting substitutions.

Definition 4.8. *(Determinism-Respecting Pairs of Substitutions)* Given a set L of labels, a pair of substitutions (σ, σ') is determinism-respecting w.r.t. a pair of sets of formulae (Φ, Φ') and L when for all two positive formulae $s \xrightarrow{l} s' \in \Phi$ and $t \xrightarrow{l} t' \in \Phi'$ such that $l \in L$, $\sigma(s) \equiv \sigma'(t)$ only if $\sigma(s') \equiv \sigma'(t')$.

We now have all the necessary ingredients to define a rule format that guarantees determinism.

Definition 4.9. *(Determinism Format)* A TSS $\mathcal{T}$ is in the determinism format w.r.t. a set of labels L, when for each $l \in L$ the following conditions hold.

1. In each deduction rule $\frac{\Phi}{t \xrightarrow{l} t'}$, each variable $v \in vars(t')$ is source dependent via a subset of $\{ \xrightarrow{l} \mid l \in L\}$, and
2. for each pair of distinct deduction rules $\frac{\Phi_0}{t_0 \xrightarrow{l} t'_0}$ and $\frac{\Phi_1}{t_1 \xrightarrow{l} t'_1}$ and for each determinism-respecting pair of substitutions (σ, σ') w.r.t. (Φ_0, Φ_1) and L such that $\sigma(t_0) \equiv \sigma'(t_1)$, it holds that either $\sigma(t'_0) \equiv \sigma'(t'_1)$ or $\sigma(\Phi_0)$ contradicts $\sigma'(\Phi_1)$.

The first constraint in the definition above ensures that each rule in a TSS in the determinism format, with some $l \in L$ as its label of conclusion, can be used to prove at most one outgoing transition for each closed term. The second requirement guarantees that no two different rules can be used to prove two distinct l-labelled transitions for any closed term.

Theorem 4.10. Consider a TSS with (C, U) as its least three-valued stable model and a subset L of its labels. If the TSS is in the determinism format w.r.t. L, then C is deterministic for each $l \in L$.

Proof. Instead of proving that C is deterministic for each $l \in L$, we establish the following more general result. We prove that, for each $l \in L$,

$$\text{if } p \xrightarrow{l} p' \in C \cup U \text{ and } p \xrightarrow{l} p'' \in C \quad \text{then} \quad p' \equiv p''. \tag{4.1}$$

Assume the first two statements. Since $p \xrightarrow{l} p' \in C \cup U$, then there exists a provable transition rule, such that $\mathcal{T} \vdash \frac{N}{p \xrightarrow{l} p'}$, for some set N of negative formulae such that $C \vDash N$. We show the claim (4.1) by an induction on the proof structure for the transition rule $\frac{N}{p \xrightarrow{l} p'}$. Let (r) be the last deduction rule, and σ the substitution, used in the proof structure for $\frac{N}{p \xrightarrow{l} p'}$.

Similarly, since $p \xrightarrow{l} p'' \in C$, there also exists a proof structure such that $\mathcal{T} \vdash \frac{N'}{p \xrightarrow{l} p''}$ for some set N' of negative formulae such that $C \cup U \vDash N'$. Let (r′) be the last deduction rule, and σ' the substitution, used in the proof structure for $\mathcal{T} \vdash \frac{N'}{p \xrightarrow{l} p''}$.

The proof is split in two main cases, the case when both proofs are based on the same rule ((r) = (r′)) and the case when they are based on two distinct rules ((r) ≠ (r′)).

Case (r) = (r′). In this case, say the rules (r) and (r′) are both the rule $\frac{\Phi}{t \xrightarrow{l} t'}$. Obviously $\sigma(t) \equiv \sigma'(t)$ since both must be equal to p. Since $\sigma(t')$ and $\sigma'(t')$ are equal to p' and p'' respectively, to show our claim (4.1) we thus need to show that $\sigma(t') \equiv \sigma'(t')$.

We define the *distance* of a source-dependent variable as the length of the shortest backward path from the variable, via premises with a label in L, to the variables in the source of conclusion. A variable in the source of the conclusion is thus of distance 0.

By induction on its distance, we now show that any variable v, which is source-dependent via a subset of $\{\xrightarrow{l} \mid l \in L\}$, is assigned the same value by σ and σ', i.e. $\sigma(v) \equiv \sigma'(v)$. The first constraint of our rule format dictates that any variable appearing in the target t' of the rule, is source-dependent via this set. Therefore if σ and σ' agree on all such variables, they must also agree on t'.

Let v be a variable appearing in the rule, which is source-dependent via some subset of $\{\xrightarrow{l} \mid l \in L\}$. The base case of the induction is simple: If the distance of v is zero, this means v appears in the source t. Since $\sigma(t) \equiv \sigma'(t)$ it must be the case that $\sigma(v) \equiv \sigma'(v)$.

For the inductive step, assume v has a non-zero distance. According to Definition 4.7 this means that v appears in the target of some premise $t_i \xrightarrow{l_i} t_i' \in \Phi$ where $l_i \in L$, and all variables appearing in t_i are also source dependent via the set $\{\xrightarrow{l} \mid l \in L\}$. However, each variable $w \in vars(t_i)$ has a smaller distance than v. By the induction hypothesis (on variable distances), we thus have that $\sigma(w) \equiv \sigma'(w)$.

At this point, we can invoke the outer induction hypothesis, namely that of our induction on the proof structure of $p \xrightarrow{l} p'$. Since $t_i \xrightarrow{l_i} t_i'$ is a premise of the first rule used, it must be provable with a smaller proof structure, using either the substitution σ or σ'. By the induction hypothesis, the claim (4.1) holds for it. In other words the target of the premise is the same whether we use σ or σ', i.e. $\sigma(t_i') \equiv \sigma'(t_i')$. Since the variable v appears in t_i', it must thus hold that $\sigma(v) \equiv \sigma'(v)$.

We have thus showed that σ and σ' agree on the value of v in all cases. As noted above, this holds specifically for all variables of t' and we can conclude that $\sigma(t') \equiv \sigma'(t')$, or $p' \equiv p''$, which proves the claim (4.1) in the case of $(\mathrm{r}) = (\mathrm{r}')$.

Case $(\mathrm{r}) \neq (\mathrm{r}')$. We now consider the case where the rules (r) and (r') are distinct. Let $(\mathrm{r}) = \frac{\Phi}{s \xrightarrow{l} s'}$ and $(\mathrm{r}') = \frac{\Phi'}{t \xrightarrow{l} t'}$. We first show that the pair (σ, σ') is determinism-respecting w.r.t. (Φ, Φ') and L.

Assume, towards a contradiction, that the pair is not determinism-respecting. Then there exist two positive formulae $s_i \xrightarrow{l'} s_i'$ and $t_i \xrightarrow{l'} t_i'$ for some $l' \in L$ among the premises of (r) and(r'), respectively, such that $\sigma(s_i) \equiv \sigma'(t_i)$ but $\sigma(s_i') \not\equiv \sigma'(t_i')$. Since $s_i \xrightarrow{l} s_i'$ is a premise of (r), we know that $\sigma(s_i \xrightarrow{l} s_i') \in C \cup U$ and it has a smaller proof structure than $p \xrightarrow{l} p' \in C \cup U$. Following a similar reasoning, $\sigma'(t_i \xrightarrow{l} t_i') \in C$. But the induction hypothesis (on the proof structure) applies and hence, we have $\sigma(s_i') \equiv \sigma'(t_i')$, which contradicts our earlier conclusion that $\sigma(s_i') \not\equiv \sigma'(t_i')$ does not hold. Hence, we conclude that our assumption is false and that (σ, σ') is determinism-respecting w.r.t. (Φ, Φ') and L.

Since we have shown that (σ, σ') is determinism respecting, it then follows from the second condition of the determinism format that either $\sigma(s') \equiv \sigma'(t')$, which was to be shown, or there exist premises $\varphi_i \equiv u_i \xrightarrow{l_i} u'_i$ in one deduction rule and $\varphi'_i \equiv w_i \overset{l_i}{\nrightarrow}$ in the other deduction rule such that $\sigma(\varphi_i)$ contradicts $\sigma'(\varphi'_i)$. We show that the latter possibility leads to a contradiction, thus completing the proof. Assume $\sigma(\varphi_i)$ contradicts $\sigma'(\varphi'_i)$, then we have that $\sigma(u_i) \equiv \sigma'(w_i)$. We distinguish the following two cases based on the status of the positive and negative contradicting premises with respect to (r) and (r′).

1. Assume that the positive formula is a premise of (r). Then, $\sigma(u_i \xrightarrow{l_i} u'_i) \in C \cup U$ but from $C \cup U \vDash N'$ and $\sigma'(\varphi'_i) \in N'$, it follows that for no p'', we have that $\sigma(u_i) \equiv \sigma'(w_i) \xrightarrow{l_i} p'' \in C \cup U$, thus reaching a contradiction.
2. Assume that the positive formula is a premise of (r′). Then, $\sigma'(u_i) \xrightarrow{l_i} \sigma(u'_i) \in C$ but from $C \vDash N$ and $\sigma(\varphi'_i) \in N$, it follows that for no p_1, we have that $\sigma(w_i) \equiv \sigma'(u_i) \xrightarrow{l_i} p_1 \in C$, hence reaching a contradiction. □

For a TSS in the determinism format with (C, U) as its least three-valued stable model, U and thus $C \cup U$ need not be deterministic. The following counter-example illustrates this phenomenon.

Example 4.11. Consider the TSS given by the following deduction rules.

$$\frac{a \xrightarrow{l} a}{a \xrightarrow{l} b} \qquad \frac{a \overset{l}{\nrightarrow}}{a \xrightarrow{l} a}$$

The above-given TSS is in the determinism format since $a \xrightarrow{l} a$ and $a \overset{l}{\nrightarrow}$ contradict each other (under any substitution). Its least three-valued stable model is, however, $(\{\}, \{a \xrightarrow{l} a, a \xrightarrow{l} b\})$ and $\{a \xrightarrow{l} a, a \xrightarrow{l} b\}$ is not deterministic.

Corollary 4.12. Consider a complete TSS with L as a subset of its labels. If the TSS is in the determinism format w.r.t. L, then its defined transition relation is deterministic for each $l \in L$.

Constraint 2 in Definition 4.9 may seem difficult to verify, since it requires checks for all possible (determinism-respecting) substitutions. However, in practical cases, to be quoted in the remainder of this chapter, variable names are chosen in such a way that constraint 2 can be checked syntactically. For example, consider the following two deduction rules.

$$\frac{x \xrightarrow{a} x'}{f(x, y) \xrightarrow{a} x'} \qquad \frac{y \overset{a}{\nrightarrow} \quad x \xrightarrow{b} x'}{f(y, x) \xrightarrow{a} x'}$$

If in both deduction rules $f(x, y)$ (or symmetrically $f(y, x)$) was used, it could have been easily seen from the syntax of the rules that the premises of one deduction rule always (under all pairs of substitutions agreeing on the value of x) contradict the premises of the other deduction rule and, hence, constraint 2 is trivially satisfied. Based on this observation, we next present a rule format, whose constraints have a purely syntactic form, and that is sufficiently powerful to handle all the examples we discuss in Section 4.3.1. (Note that, for the examples in Section 4.3.1, checking the constraints of Definition 4.9 is not too hard either.)

Definition 4.13. *(Normalized TSSs)* A TSS is normalized w.r.t. L if each deduction rule is f-defining for some function symbol f, and for each label $l \in L$, each function symbol f and each pair of deduction rules of the form

$$(\mathrm{r})\ \frac{\Phi_r}{f(\overrightarrow{s}) \xrightarrow{l} s'} \qquad (\mathrm{r}')\ \frac{\Phi_{r'}}{f(\overrightarrow{t}) \xrightarrow{l} t'}$$

the following constraints are satisfied:

1. the sources of the conclusions coincide, i.e., $f(\overrightarrow{s}) \equiv f(\overrightarrow{t})$,
2. each variable $v \in vars(s')$ (symmetrically $v \in vars(t')$) is source dependent in (r) (respectively in (r′)) via some subset of $\{\xrightarrow{l} \mid l \in L\}$,
3. for each variable $v \in vars(r) \cap vars(r')$ there is a set of formulae in $\Phi_r \cap \Phi_{r'}$ proving its source dependency (both in (r) and (r′))) via some subset of $\{\xrightarrow{l} \mid l \in L\}$.

The second and third constraint in Definition 4.14 guarantee that the syntactic equivalence of relevant terms (the target of the conclusion or the premises negating each other) will lead to syntactically equivalent closed terms under all determinism-respecting pairs of substitutions.

The reader can check that all the examples quoted from the literature in Section 4.3.1 are indeed normalized TSSs.

Definition 4.14. *(Syntactic Determinism Format)* A normalized TSS is in the (syntactic) determinism format w.r.t. L, when for each two deduction rules $\frac{\Phi_0}{f(\overrightarrow{s}) \xrightarrow{l} s'}$ and $\frac{\Phi_1}{f(\overrightarrow{s}) \xrightarrow{l} s''}$, it holds that $s' \equiv s''$ or Φ_0 contradicts Φ_1.

The following theorem states that for normalized TSSs, Definition 4.14 implies Definition 4.9.

Theorem 4.15. Each normalized TSS in the syntactic determinism format w.r.t. L is also in the determinism format w.r.t. L.

Proof. Let $\mathcal{T}$ be a normalized TSS in the syntactic determinism format w.r.t. L. Condition 1 of Definition 4.9 is satisfied since $\mathcal{T}$ is normalized. To see this, consider item 2 of Definition 4.13, by taking (r) and (r′) to be the same rule.

To prove condition 2 of Definition 4.9 let $(r) = \frac{\Phi_0}{t_0 \xrightarrow{l} t_0'}$ and $(r') = \frac{\Phi_1}{t_1 \xrightarrow{l} t_1'}$ be distinct rules of $\mathcal{T}$ and (σ, σ') be a determinism-respecting pair of substitutions w.r.t. (Φ_0, Φ_1) and L such that $\sigma(t_0) \equiv \sigma'(t_1)$. Since $\mathcal{T}$ is normalized, both (r) and (r′) are f-defining for some function symbol f, i.e., $t_0 = f(\overrightarrow{s})$ and $t_1 = f(\overrightarrow{t})$.

Furthermore, since $f(\overrightarrow{s}) \equiv f(\overrightarrow{t})$ we have that σ and σ' agree on all variables appearing in $f(\overrightarrow{s}) = f(\overrightarrow{t})$.

For each variable $v \in vars(r) \cap vars(r')$, we define its *common source distance* to be the source distance of v when only taking the formulae in $\Phi_0 \cap \Phi_1$ into account. Note that such a source distance exists since by constraint 3 of Definition 4.13 all $v \in vars(r) \cap vars(r')$ are source dependent via a subset of $\{ \xrightarrow{l} \mid l \in L\}$ included in $\Phi_0 \cap \Phi_1$.

We prove for each $v \in vars(r) \cap vars(r')$ that $\sigma(v) \equiv \sigma'(v)$ by an induction on the common source distance of variables v. Suppose that we show the above claim, then we can prove the theorem as follows. It follows from Definition 4.14 that either $t'_0 \equiv t'_1$ or Φ_0 contradicts Φ_1. If $t'_0 \equiv t'_1$, then variables in $vars(t'_0) = vars(t'_1)$ are all source dependent via transitions in L that are common to both Φ_0 and Φ_1 (by constraint 3 of Definition 4.13). By the above-mentioned claim, $\sigma(t'_0) \equiv \sigma'(t'_1)$, thus, constraint 2 of Definition 4.9 follows, which was to be shown. If Φ_0 contradicts Φ_1, then assume that the premises negating each other are $\varphi_j \equiv s_j \xrightarrow{l_j} s'_j$ and $\varphi_{j'} \equiv t_{j'} \overset{l_j}{\nrightarrow}$ and it holds that $s_j \equiv t_{j'}$. All variables in $t_j \equiv s_j$ are source dependent via transitions in L (by constraint 3 of Definition 4.13). It follows from the claim that $\sigma(s_j) \equiv \sigma'(t_{j'})$ and thus, $\sigma(\varphi_j)$ contradicts $\sigma'(\varphi_{j'})$, which implies constraint 2 of Definition 4.9.

Hence, it only remains to prove, by an induction on the common source distance of v, that $\sigma(v) \equiv \sigma'(v)$. If $v \in vars(f(\overrightarrow{s}))$ then we know that $\sigma(v) \equiv \sigma'(v)$ (since $t_0 \equiv t_1$ and $\sigma(t_0) \equiv \sigma'(t_1)$. Otherwise, since v is source dependent in (r) via transitions with labels in L, there is a positive premise $u \xrightarrow{l} u'$ in Φ_0 with $l \in L$ such that $v \in vars(u')$ and all variables in u are source dependent with a shorter common source distance. Furthermore, since v appears in both rules, i.e., $v \in vars(r) \cap vars(r')$, this premise also appears in Φ_1 according to item 3 of Definition 4.13 and thus $vars(u) \subseteq vars(r) \cap vars(r')$. By the induction hypothesis we have that $\sigma(u) \equiv \sigma'(u)$ and since (σ, σ') is determinism-respecting w.r.t. (Φ_0, Φ_1) and L, we know that $\sigma(u') \equiv \sigma'(u')$. Specifically, the substitutions must agree on the value of v, i.e. $\sigma(v) \equiv \sigma'(v)$. ☐

The following statement is a corollary to Theorems 4.15 and 4.10.

Corollary 4.16. Consider a normalized TSS with (C, U) as its least three-valued stable model and a subset L of its labels. If the TSS is in the (syntactic) determinism format w.r.t. L (according to Definition 4.14), then C is deterministic w.r.t. any $l \in L$.

4.3.1 Examples

In this section, we present some examples of various TSSs from the literature and apply our (syntactic) determinism format to them. Some of the examples we discuss below are based on TSSs with predicates. The extension of our formats with predicates is straightforward and we discuss it in Section 4.4.3 to follow.

Example 4.17. *(Conjunctive Nondeterministic Processes)* Hennessy and Plotkin, in [Hennessy and Plotkin 1987], define a language, called conjunctive nondeterministic processes, for studying logical characterizations of processes. The signature of the language consists of a constant 0, a unary action prefixing operator $a.\_$ for each $a \in A$, and a binary conjunctive nondeterminism operator $\vee$. The operational semantics of this language is defined by the following deduction rules.

$$\frac{}{0\ \mathrm{can}_a} \qquad \frac{}{a.x\ \mathrm{can}_a} \qquad \frac{x\ \mathrm{can}_a}{x \vee y\ \mathrm{can}_a} \qquad \frac{y\ \mathrm{can}_a}{x \vee y\ \mathrm{can}_a}$$

$$\frac{}{0\ \mathrm{after}_a\ 0} \qquad \frac{}{a.x\ \mathrm{after}_a\ x} \qquad \frac{}{a.x\ \mathrm{after}_b\ 0}\ a \neq b \qquad \frac{x\ \mathrm{after}_a\ x' \quad y\ \mathrm{after}_a\ y'}{x \vee y\ \mathrm{after}_a\ x' \vee y'}$$

The above TSS is in the (syntactic) determinism format with respect to the transition relation after_a (for each $a \in A$). Hence, we can conclude that the transition relations after_a are deterministic.

Example 4.18. *(Delayed choice)* The second example we discuss is a subset of the process algebra $\mathrm{BPA}_{\delta\epsilon}$ + DC [Baeten and Mauw 1995], i.e., Basic Process Algebra with deadlock and empty process extended with delayed choice. First we restrict attention to the fragment of this process algebra without non-deterministic choice $+$ and with action prefix $a.\_$ instead of general sequential composition $\cdot$. This altered process algebra has the following deduction rules, where a ranges over the set of actions A:

$$\frac{}{\epsilon\downarrow} \qquad \frac{}{a.x \xrightarrow{a} x} \qquad \frac{x\downarrow}{x \mp y\downarrow} \qquad \frac{y\downarrow}{x \mp y\downarrow}$$

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'} \qquad \frac{x \xrightarrow{a} x' \quad y \stackrel{a}{\not\rightarrow}}{x \mp y \xrightarrow{a} x'} \qquad \frac{x \stackrel{a}{\not\rightarrow} \quad y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'}$$

In the above specification, predicate $p\downarrow$ denotes the possibility of termination for process p. The intuition behind the delayed choice operator, denoted by $\_ \mp \_$, is that the choice between two components is only resolved when one performs an action that the other cannot perform. When both components can perform an action, the delayed choice between them remains unresolved and the two components synchronize on the common action. This transition system specification is in the (syntactic) determinism format w.r.t. $\{a \mid a \in A\}$.

Addition of non-deterministic choice $+$ or sequential composition $\cdot$ results in deduction rules that do not satisfy the determinism format. For example, addition of sequential composition comes with the following deduction rules:

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x\downarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$$

The sets of premises of these rules do not contradict each other. The extended TSS is indeed non-deterministic since, for example, $(\epsilon \mp (a.\epsilon)) \cdot (a.\epsilon) \xrightarrow{a} \epsilon$ and $(\epsilon \mp (a.\epsilon)) \cdot (a.\epsilon) \xrightarrow{a} \epsilon \cdot (a.\epsilon)$.

Example 4.19. *(Time transitions I)* This example deals with the Algebra of Timed Processes, ATP, of Nicollin and Sifakis [Nicollin and Sifakis 1994]. In the TSS given below, we specify the time transitions (denoted by label χ) of delayable deadlock δ, non-deterministic choice $\_ \oplus \_$, unit-delay operator $\lfloor \_ \rfloor \_$ and parallel composition $\_ \parallel \_$.

$$\frac{}{\delta \xrightarrow{\chi} \delta} \qquad \frac{x \xrightarrow{\chi} x' \quad y \xrightarrow{\chi} y'}{x \oplus y \xrightarrow{\chi} x' \oplus y'} \qquad \frac{}{\lfloor x \rfloor (y) \xrightarrow{\chi} y} \qquad \frac{x \xrightarrow{\chi} x' \quad y \xrightarrow{\chi} y'}{x \parallel y \xrightarrow{\chi} x' \parallel y'}$$

These deduction rules all trivially satisfy the determinism format for time transitions since the sources of conclusions of different deduction rules cannot be unified. Also the additional operators involving time, namely, the delay operator $\lfloor \_ \rfloor^d \_$, execution delay operator $\lceil \_ \rceil^d \_$ and unbounded start delay operator $\lfloor \_ \rfloor^\omega$, satisfy the determinism format for time transitions. The deduction rules are given below, for $d \geq 1$:

$$\frac{}{\lfloor x \rfloor^1 (y) \xrightarrow{\chi} y} \qquad \frac{x \xrightarrow{\chi} x'}{\lfloor x \rfloor^{d+1} (y) \xrightarrow{\chi} \lfloor x' \rfloor^d (y)} \qquad \frac{x \overset{\chi}{\not\rightarrow}}{\lfloor x \rfloor^{d+1} (y) \xrightarrow{\chi} \lfloor x \rfloor^d (y)}$$

$$\frac{x \xrightarrow{\chi} x'}{\lfloor x \rfloor^\omega \xrightarrow{\chi} \lfloor x' \rfloor^\omega} \qquad \frac{x \overset{\chi}{\not\rightarrow}}{\lfloor x \rfloor^\omega \xrightarrow{\chi} \lfloor x \rfloor^\omega}$$

$$\frac{x \xrightarrow{\chi} x'}{\lceil x \rceil^1 (y) \xrightarrow{\chi} y} \qquad \frac{x \xrightarrow{\chi} x'}{\lceil x \rceil^{d+1} (y) \xrightarrow{\chi} \lceil x' \rceil^d (y)}$$

Example 4.20. *(Time transitions II)* Most of the timed process algebras that originate from the Algebra of Communicating Processes (ACP) from [Bergstra and Klop 1984, Baeten and Weijland 1990] such as those reported in [Baeten and Middelburg 2002] have a deterministic time transition relation as well.

In the TSS given below, the time unit delay operator is denoted by $\sigma_{\text{rel}}\_$, nondeterministic choice is denoted by $\_ + \_$, and sequential composition is denoted by $\_ \cdot \_$. The deduction rules for the time transition relation for this process algebra are the following:

$$\frac{}{\sigma_{\text{rel}}(x) \xrightarrow{1} x} \qquad \frac{x \xrightarrow{1} x' \quad y \xrightarrow{1} y'}{x + y \xrightarrow{1} x' + y'} \qquad \frac{x \xrightarrow{1} x' \quad y \overset{1}{\not\rightarrow}}{x + y \xrightarrow{1} x'} \qquad \frac{x \overset{1}{\not\rightarrow} \quad y \xrightarrow{1} y'}{x + y \xrightarrow{1} y'}$$

$$\frac{x \xrightarrow{1} x' \quad x \not\downarrow}{x \cdot y \xrightarrow{1} x' \cdot y} \qquad \frac{x \xrightarrow{1} x' \quad y \overset{1}{\not\rightarrow}}{x \cdot y \xrightarrow{1} x' \cdot y} \qquad \frac{x \xrightarrow{1} x' \quad x \downarrow \quad y \xrightarrow{1} y'}{x \cdot y \xrightarrow{1} x' \cdot y + y'}$$

$$\frac{x \overset{1}{\not\rightarrow} \quad x \downarrow \quad y \xrightarrow{1} y'}{x \cdot y \xrightarrow{1} y'}$$

Note that here we have an example of deduction rules, the first two deduction rules for time transitions of a sequential composition, for which the premises do not contradict each other. Still these deduction rules satisfy the determinism format since the targets of the conclusions are identical. In the syntactically richer framework of [van Weerdenburg and Reniers 2008], where arbitrary first-order logic formulae over transitions are allowed, those two deduction rules are presented by a single rule with premise $x \xrightarrow{1} x' \wedge (x \not\downarrow \vee y \overset{1}{\nrightarrow})$.

Sometimes such timed process algebras have an operator for specifying an arbitrary delay, denoted by $\sigma^*_{\mathrm{rel}\_}$, with the following deduction rules.

$$\frac{x \overset{1}{\nrightarrow}}{\sigma^*_{\mathrm{rel}}(x) \xrightarrow{1} \sigma^*_{\mathrm{rel}}(x)} \qquad \frac{x \xrightarrow{1} x'}{\sigma^*_{\mathrm{rel}}(x) \xrightarrow{1} x' + \sigma^*_{\mathrm{rel}}(x)}$$

The premises of these rules contradict each other and so, the semantics of this operator also satisfies the constraints of our (syntactic) determinism format.

4.4 Idempotency

Our order of business in this section is to present a rule format that guarantees the idempotency of certain binary operators. In the definition of our rule format, we rely implicitly on the work presented in the previous section.

4.4.1 Format

Definition 4.21. *(Idempotency)* A binary operator $f \in \Sigma$ is *idempotent w.r.t. an equivalence* $\sim$ on closed terms if and only if for each $p \in \mathbb{C}(\Sigma)$, it holds that $f(p,p) \sim p$.

Idempotency is defined with respect to a notion of behavioral equivalence. There are various notions of behavioral equivalence defined in the literature, which are by and large, weaker than bisimilarity defined below. Thus, to be as general as possible, we prove our idempotency result for all notions that contain, i.e., are weaker than, bisimilarity.

Definition 4.22. *(Bisimulation)* Let $\mathcal{T}$ be a TSS with signature Σ. A relation $\mathcal{R} \subseteq \mathbb{C}(\Sigma) \times \mathbb{C}(\Sigma)$ is a *bisimulation relation* if and only if $\mathcal{R}$ is symmetric and for all $p_0, p_1, p'_0 \in \mathbb{C}(\Sigma)$ and $l \in L$

$$(p_0 \, \mathcal{R} \, p_1 \wedge \mathcal{T} \vdash p_0 \xrightarrow{l} p'_0) \Rightarrow \exists_{p'_1 \in \mathbb{C}(\Sigma)} (\mathcal{T} \vdash p_1 \xrightarrow{l} p'_1 \wedge p'_0 \, \mathcal{R} \, p'_1).$$

Two terms $p_0, p_1 \in \mathbb{C}(\Sigma)$ are called *bisimilar*, denoted by $p_0 \leftrightarrows p_1$, when there exists a bisimulation relation $\mathcal{R}$ such that $p_0 \mathcal{R} p_1$.

Definition 4.23. *(The Idempotency Rule Format)* Let $\gamma : L \times L \to L$ be a partial function such that $\gamma(l_0, l_1) \in \{l_0, l_1\}$ if it is defined. We define the following two rule forms.

1_l. Choice rules

$$\frac{\{x_i \xrightarrow{l} t\} \cup \Phi}{f(x_0, x_1) \xrightarrow{l} t}, \quad i \in \{0,1\}$$

$2_{l_0,l_1}$. Communication rules

$$\frac{\{x_0 \xrightarrow{l_0} t_0,\ x_1 \xrightarrow{l_1} t_1\} \cup \Phi}{f(x_0, x_1) \xrightarrow{\gamma(l_0,l_1)} f(t_0, t_1)}, \quad t_0 \equiv t_1 \text{ or } (l_0 = l_1 \text{ and } \xrightarrow{l_0} \text{ is deterministic })$$

In each case, Φ can be an arbitrary, possibly empty set of (positive or negative) formulae.

In addition, we define the starred version of each form, 1_l^* and $2_{l_0,l_1}^*$. The starred version of each rule is the same as the unstarred one except that t, t_0 and t_1 are restricted to be concrete variables and the set Φ must be empty in each case.

A TSS is in *idempotency format w.r.t. a binary operator f* if each f-defining rule, i.e., a deduction rule with f appearing in the source of the conclusion, is of the forms 1_l or $2_{l_0,l_1}$, for some $l, l_0, l_1 \in L$, and for each label $l \in L$ there exists at least one rule of the forms 1_l^* or $2_{l,l}^*$.

We should note that the starred versions of the forms are special cases of their unstarred counterparts; for example a rule which has form 1_l^* also has form 1_l.

Theorem 4.24. Assume that a TSS is complete and is in the idempotency format with respect to a binary operator f. Then, f is idempotent w.r.t. to any equivalence $\sim$ such that $\underline{\leftrightarrow} \subseteq \sim$.

Proof. First define the relation $\simeq_f \subseteq \mathbb{C}(\Sigma) \times \mathbb{C}(\Sigma)$ as follows.

$$\simeq_f = \{(p,p), (p, f(p,p)), (f(p,p), p) \mid p \in \mathbb{C}(\Sigma)\}$$

To prove the theorem it suffices to show that $\simeq_f$ is a bisimulation relation. If it is, then $f(p,p) \underline{\leftrightarrow}\, p$ for any closed term p and since $\underline{\leftrightarrow} \subseteq \sim$ we obtain the theorem.

Let (C, U) be the least three-valued stable model for the TSS under consideration. First consider a closed term p s.t. $p \xrightarrow{l} p' \in C$ for some l and p' (note that $U = \{\}$ since the TSS is complete). Next, we argue that $f(p,p) \xrightarrow{l} p''$ for some p'' such that $p' \simeq_f p''$. Since $p \xrightarrow{l} p' \in C$, there exists a provable transition rule of the form $\frac{N}{p \xrightarrow{l} p'}$ for some set of negative formulae N such that $C \vDash N$. In particular, that means that $p \overset{l}{\nrightarrow} \notin N$. In this case we make use of the requirement that there exists at least one rule of a starred form for label l. If there exists a rule of the form 1_l^*, i.e.

$$\frac{x_i \xrightarrow{l} x'}{f(x_0, x_1) \xrightarrow{l} x'}, \quad i \in \{0,1\}$$

then we can instantiate it to prove that $f(p,p) \xrightarrow{l} p' \in C$. In particular, it does not matter if $i = 0$ or $i = 1$. Since $\simeq_f$ is reflexive, $p' \simeq_f p'$ holds. If there exists a rule of the form $2^*_{l,l}$, we observe that $\gamma(l,l) = l$ so the transition rule becomes

$$\frac{x_0 \xrightarrow{l} x_0' \quad x_1 \xrightarrow{l} x_1'}{f(x_0,x_1) \xrightarrow{l} f(x_0',x_1')} \ ,$$

where $x_0' \equiv x_1'$ or $\xrightarrow{l}$ is deterministic. Now we can use the existence of $p \xrightarrow{l} p'$ to satisfy both premises and obtain that $f(p,p) \xrightarrow{l} f(p',p')$. By the definition of $\simeq_f$ we also have that $p' \simeq_f f(p',p')$. In either case, if $p \xrightarrow{l} p' \in C$, then there exists a p'' s.t. $f(p,p) \xrightarrow{l} p'' \in C$ and $p' \simeq_f p''$.

Now assume that $f(p,p) \xrightarrow{k} p' \in C$. Then there exists a provable transition rule $\frac{N}{f(p,p) \xrightarrow{k} p'}$ for some set of negative formulae N such that $C \vDash N$. Since all rules for f are either of the form 1_l or $2_{l_0,l_1}$, this provable transition rule must be based on a rule of those forms. We analyze each possibility separately, showing that in each case $p \xrightarrow{k} p''$ for some p'' such that $p' \simeq_f p''$.

If the rule is based on a rule of form 1_l, its positive premises must also be provable. In particular it must hold that $p \xrightarrow{k} p' \in C$ since both x_0 and x_1 in the rule are instantiated to p. The other premises are of no consequence to this conclusion and, again, we observe that $p' \simeq_f p'$.

Now consider the case where the transition is a consequence of a rule of the form $2_{l_0,l_1}$. If $t_0 \equiv t_1$, say both are equal to p'', we must consider two cases, namely $k = l_0$ and $k = l_1$. If $k = l_0$ then the first premise of the rule actually states that $p \xrightarrow{k} p''$. If $k = l_1$ then the second premise similarly states that $p \xrightarrow{k} p''$. In either case, we note that $p' \equiv f(p'',p'')$ must hold and again by the definition of $\simeq_f$ we have that $f(p'',p'') \simeq_f p''$.

If however $t_0 \not\equiv t_1$ the side condition requires that $l_0 = l_1 = k$, which also implies $\gamma(l_0,l_1) = l_0 = k$, and that the transition relation $\xrightarrow{l_0}$ is deterministic. In this case it is easy to see that the right-hand sides of the first two premises, namely t_0 and t_1, evaluate to the same closed term in the proof structure, say p''. The conclusion then states that $k = l_0$ and $p' \equiv f(p'',p'')$. It must thus hold that $p \xrightarrow{k} p'' \in C$ and $f(p'',p'') \simeq_f p''$ as before.

From this we obtain that if $f(p,p) \xrightarrow{k} p' \in C$ then there exists a p'' such that $p \xrightarrow{k} p'' \in C$ and $p' \simeq_f p''$. Thus, $\simeq_f$ is a bisimulation. □

4.4.2 Relaxing the restrictions

In this section we consider the constraints of the idempotency rule format (Definition 4.23) and show that they cannot be dropped without jeopardizing the meta-theorem.

First of all we note that, in rule form 1_l, it is necessary that the label of the premise matches the label of the conclusion. If it does not, in general, we cannot prove that $f(p,p)$ simulates p or vice versa. This requirement can be stated more generally for both rule forms in Definition 4.23; the label of the conclusion must be among the labels of the premises. The requirement that $\gamma(l,l') \in \{l,l'\}$ exists to ensure this constraint for form $2_{l,l'}$. A simple synchronization rule provides a counter-example that shows why this restriction is needed. Consider the following TSS with constants 0, τ, a and $\bar{a}$ and two binary operators + and $\|$.

$$\frac{}{\alpha \xrightarrow{\alpha} 0} \qquad \frac{x \xrightarrow{\alpha} x'}{x + y \xrightarrow{\alpha} x'} \qquad \frac{y \xrightarrow{\alpha} y'}{x + y \xrightarrow{\alpha} y'} \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{\bar{a}} y'}{x \parallel y \xrightarrow{\tau} x' \parallel y'}$$

where α is τ, a or $\bar{a}$. Here it is easy to see that although $(a + \bar{a}) \parallel (a + \bar{a})$ has an outgoing τ-transition, $a + \bar{a}$ does not afford such a transition.

The condition that for each l at least one rule of the forms 1^*_l or $2^*_{l,l}$ must exist comprises a few constraints on the rule format. First of all, it says there must be at least one f-defining rule. If not, it is easy to see that there could exist a process p where $f(p,p)$ deadlocks (since there are no f-defining rules) but p does not. It also states that there must be at least one rule in the starred form, where the targets are restricted to variables. To motivate these constraints, consider the following TSS.

$$\frac{}{a \xrightarrow{a} 0} \qquad \frac{x \xrightarrow{a} a}{f(x,y) \xrightarrow{a} a}$$

The processes a and $f(a,a)$ are not bisimilar as the former can do an a-transition but the latter is stuck. The starred forms also require that Φ is empty, i.e. there is no testing. This is necessary in the proof because in the presence of extra premises, we cannot in general instantiate such a rule to show that $f(p,p)$ simulates p. Finally, the condition requires that if we rely on a rule of the form $2^*_{l,l'}$ and $t_0 \not\equiv t_1$, then the labels l and l' in the premises of the rule must coincide. To see why, consider a TSS containing a *left synchronize* operator $\rrbracket$, one that synchronizes a step from each operand but uses the label of the left one. Here we let $\alpha \in \{a, \bar{a}\}$.

$$\frac{}{\alpha \xrightarrow{\alpha} 0} \qquad \frac{x \xrightarrow{\alpha} x'}{x + y \xrightarrow{\alpha} x'} \qquad \frac{y \xrightarrow{\alpha} y'}{x + y \xrightarrow{\alpha} y'} \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{\bar{a}} y'}{x \rrbracket y \xrightarrow{a} x' \rrbracket y'}$$

In this TSS the processes $(a + \bar{a})$ and $(a + \bar{a}) \rrbracket (a + \bar{a})$ are not bisimilar since the latter does not afford an $\bar{a}$-transition whereas the former does.

For rules of form $2_{l,l'}$ we require that either $t_0 \equiv t_1$, or that the mentioned labels are the same and the associated transition relation is deterministic. This requirement is necessary in the proof to ensure that the target of the conclusion fits our definition of $\simeq_f$, i.e. the operator is applied to two identical terms. Consider the following TSS where $\alpha \in \{a, b\}$.

$$\frac{}{a \xrightarrow{a} a} \qquad \frac{}{a \xrightarrow{a} b} \qquad \frac{}{b \xrightarrow{b} b} \qquad \frac{x \xrightarrow{\alpha} x' \quad y \xrightarrow{\alpha} y'}{x|y \xrightarrow{\alpha} x'|y'}$$

For the operator $|$, this violates the condition $t_0 \equiv t_1$ (note that $\xrightarrow{a}$ is not deterministic). We observe that $a|a \xrightarrow{a} a|b$. The only possibilities for a to simulate this

a-transition is either with $a \xrightarrow{a} a$ or with $a \xrightarrow{a} b$. However, neither a nor b can be bisimilar to $a|b$ because both a and b have outgoing transitions while $a|b$ is stuck. Therefore a and $a|a$ cannot be bisimilar. If $t_0 \not\equiv t_1$ we must require that the labels match, $l_0 = l_1$, and that $\xrightarrow{l_0}$ is deterministic. We require the labels to match because if they do not, then given only $p \xrightarrow{l} p'$ it is impossible to prove that $f(p,p)$ can simulate it using only a $2^*_{l,l'}$ rule. The determinacy of the transition with that label is necessary when proving that transitions from $f(p,p)$ can, in general, be simulated by p; if we assume that $f(p,p) \xrightarrow{l} p'$ then we must be able to conclude that p' has the shape $f(p'',p'')$ for some p'', in order to meet the bisimulation condition for $\simeq_f$. Consider the standard choice operator + and prefixing operator . of CCS with the | operator from the last example, with $\alpha \in \{a,b,c\}$.

$$\frac{}{\alpha \xrightarrow{\alpha} 0} \qquad \frac{}{\alpha.x \xrightarrow{\alpha} x} \qquad \frac{x \xrightarrow{\alpha} x'}{x + y \xrightarrow{\alpha} x'} \qquad \frac{y \xrightarrow{\alpha} y'}{x + y \xrightarrow{\alpha} y'} \qquad \frac{x \xrightarrow{\alpha} x' \quad y \xrightarrow{\alpha} y'}{x|y \xrightarrow{\alpha} x'|y'}$$

If we let $p = a.b + a.c$, then $p|p \xrightarrow{a} b|c$ and $b|c$ is stuck. However, p cannot simulate this transition w.r.t. $\simeq_f$. Indeed, p and $p|p$ are not bisimilar.

4.4.3 Predicates

There are many examples of TSSs where predicates are used. The definitions presented in Section 4.2 and 4.4 can be easily adapted to deal with predicates as well. In particular, two closed terms are called bisimilar in this setting when, in addition to the transfer conditions of bisimilarity, they satisfy the same predicates. To extend the idempotency rule format to a setting with predicates, the following types of rules for predicates are introduced:

3_P. **Choice rules for predicates**

$$\frac{\{Px_i\} \cup \Phi}{Pf(x_0,x_1)}, \quad i \in \{0,1\}$$

4_P. **Synchronization rules for predicates**

$$\frac{\{Px_0,\ Px_1\} \cup \Phi}{Pf(x_0,x_1)}$$

As before, we define the starred version of these forms, 3^*_P and 4^*_P. The starred version of each rule is the same as the unstarred one except that the set Φ must be empty in each case. With these additional definition the idempotency format is defined as follows.

A TSS is in *idempotency format w.r.t. a binary operator f* if each f-defining rule, i.e., a deduction rule with f appearing in the source of the conclusion, is of one the forms 1_l, $2_{l_0,l_1}$, 3_P or 4_P for some $l, l_0, l_1 \in L$, for each label $l \in L$ and predicate symbol P. Moreover, for each $l \in L$, there exists at least one rule of the forms 1^*_l or $2^*_{l,l}$, and for each predicate symbol P there is a rule of the form 1^*_P or 2^*_P.

4.4.4 Examples

Example 4.25. The most prominent example of an idempotent operator is non-deterministic choice, denoted +. It typically has the following deduction rules:

$$\frac{x_0 \xrightarrow{a} x_0'}{x_0 + x_1 \xrightarrow{a} x_0'} \qquad \frac{x_1 \xrightarrow{a} x_1'}{x_0 + x_1 \xrightarrow{a} x_1'}$$

Clearly, these are in the idempotency format w.r.t. +.

Example 4.26. *(External choice)* The well-known external choice operator $\Box$ from CSP [Hoare 1985] has the following deduction rules

$$\frac{x_0 \xrightarrow{a} x_0'}{x_0 \Box x_1 \xrightarrow{a} x_0'} \qquad \frac{x_1 \xrightarrow{a} x_1'}{x_0 \Box x_1 \xrightarrow{a} x_1'} \qquad \frac{x_0 \xrightarrow{\tau} x_0'}{x_0 \Box x_1 \xrightarrow{\tau} x_0' \Box x_1} \qquad \frac{x_1 \xrightarrow{\tau} x_1'}{x_0 \Box x_1 \xrightarrow{\tau} x_0 \Box x_1'}$$

Note that the third and fourth deduction rule are not instances of any of the allowed types of deduction rules. Therefore, no conclusion about the validity of idempotency can be drawn from our format. In this case this does not point to a limitation of our format, because this operator is not idempotent in strong bisimulation semantics [D'Argenio 1995].

Example 4.27. *(Strong time-deterministic choice)* The choice operator that is used in the timed process algebra ATP [Nicollin and Sifakis 1994] has the following deduction rules.

$$\frac{x_0 \xrightarrow{a} x_0'}{x_0 \oplus x_1 \xrightarrow{a} x_0'} \qquad \frac{x_1 \xrightarrow{a} x_1'}{x_0 \oplus x_1 \xrightarrow{a} x_1'} \qquad \frac{x_0 \xrightarrow{\chi} x_0' \quad x_1 \xrightarrow{\chi} x_1'}{x_0 \oplus x_1 \xrightarrow{\chi} x_0' \oplus x_1'}$$

The idempotency of this operator follows from our format since the last rule for $\oplus$ fits the form $2^*_{\chi,\chi}$ because, as we remarked in Example 4.19, the transition relation $\xrightarrow{\chi}$ is deterministic.

Example 4.28. *(Weak time-deterministic choice)* The choice operator + that is used in most ACP-style timed process algebras [Baeten and Middelburg 2002] has the following deduction rules:

$$\frac{x_0 \xrightarrow{a} x_0'}{x_0 + x_1 \xrightarrow{a} x_0'} \qquad \frac{x_1 \xrightarrow{a} x_1'}{x_0 + x_1 \xrightarrow{a} x_1'}$$

$$\frac{x_0 \xrightarrow{1} x_0' \quad x_1 \xrightarrow{1} x_1'}{x_0 + x_1 \xrightarrow{1} x_0' + x_1'} \qquad \frac{x_0 \xrightarrow{1} x_0' \quad x_1 \not\xrightarrow{1}}{x_0 + x_1 \xrightarrow{1} x_0'} \qquad \frac{x_0 \not\xrightarrow{1} \quad x_1 \xrightarrow{1} x_1'}{x_0 + x_1 \xrightarrow{1} x_1'}$$

The third deduction rule is of the form $2^*_{1,1}$, the others are of forms 1^*_a and 1^*_1. This operator is idempotent (since the transition relation $\xrightarrow{1}$ is deterministic, as remarked in Example 4.20).

Example 4.29. *(Conjunctive nondeterminism)* The operator $\vee$ as defined in Example 4.17 by means of the deduction rules

$$\frac{x \ \text{can}_a}{x \vee y \ \text{can}_a} \qquad \frac{y \ \text{can}_a}{x \vee y \ \text{can}_a} \qquad \frac{x \ \text{after}_a \ x' \quad y \ \text{after}_a \ y'}{x \vee y \ \text{after}_a \ x' \vee y'}$$

satisfies the idempotency format (extended to a setting with predicates). The first two deduction rules are of the form $3^*_{\text{can}_a}$ and the last one is of the form $2^*_{a,a}$. Here we have used the fact that the transition relations after_a are deterministic as concluded in Example 4.17.

Example 4.30. *(Delayed choice)* Delayed choice can be concluded to be idempotent in the restricted setting without $+$ and $\cdot$ by using the idempotency format and the fact that in this restricted setting the transition relations $\xrightarrow{a}$ are deterministic. (See Example 4.18.)

$$\frac{x \downarrow}{x \mp y \downarrow} \qquad \frac{y \downarrow}{x \mp y \downarrow} \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'}$$

$$\frac{x \xrightarrow{a} x' \quad y \not\xrightarrow{a}}{x \mp y \xrightarrow{a} x'} \qquad \frac{x \not\xrightarrow{a} \quad y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'}$$

The first two deduction rules are of form $3^*_{\downarrow}$, the third one is a $2^*_{a,a}$ rule, and the others are 1_a rules. Note that for any label a starred rule is present.

For the extensions discussed in Example 4.18 idempotency cannot be established using our rule format since the transition relations are no longer deterministic. In fact, delayed choice is not idempotent in these cases.

4.5 Conclusions

In this chapter, we presented two rule formats guaranteeing determinism of certain transitions and idempotency of binary operators. Our rule formats cover all practical cases of determinism and idempotency that we have thus far encountered in the literature.

We plan to extend our rule formats with the addition of data/store. Also, it is interesting to study the addition of structural congruences pertaining to idempotency to the TSSs in our idempotency format.

Bibliography

Martín Abadi and Cédric Fournet. Access control based on execution history. In *Networked and Distributed System Security Symposium*, 2003.

Luca Aceto, Willem Jan (Wan) Fokkink, and Chris Verhoef. Structural operational semantics. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra, Chapter 3*, pages 197–292. Elsevier Science, Dordrecht, The Netherlands, 2001.

Henrik Reif Andersen. Partial model checking (extended abstract). In *LICS*, pages 398–407, 1995.

J.C.M. (Jos) Baeten and Cornelis A. (Kees) Middelburg. *Process Algebra with Timing*. EATCS Monographs. Springer-Verlag, Berlin, Germany, 2002.

J.C.M. (Jos) Baeten and W. Peter Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambrdige University Press, 1990.

Jos C. M. Baeten and Sjouke Mauw. Delayed choice: an operator for joining Message Sequence Charts. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, Berne, Switzerland, 1994*, volume 6 of *IFIP Conference Proceedings*, pages 340–354. Chapman & Hall, 1995.

Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

Arnar Birgisson, Mohan Dhawan, Úlfar Erlingsson, Vinod Ganapathy, and Liviu Iftode. Security enforcement using software transactional memory. In *ACM Conference on Computer and Communications Security*, October 2008.

Johannes Borgström, Simon Kramer, and Uwe Nestmann. Calculus of cryptographic communication. In *Proceedings of FCS-ARSPA*, 2006.

Gérard Boudol and Ilaria Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae*, 11(4):433–452, 1988.

E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131, pages 52–71, 1981.

Sjoerd Cranen, MohammadReza Mousavi, and Michel A. Reniers. A rule format for associativity. In Franck van Breugel and Marsha Chechik, editors, *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR'08)*, volume 5201 of *Lecture Notes in Computer Science*, pages 447–461, Toronto,Canada, 2008. Springer-Verlag, Berlin, Germany.

Pedro R. D'Argenio. τ-angelic choice for process algebras (revised version). Technical report, March 1995.

Francien Dechesne, MohammadReza Mousavi, and Simona Orzan. Operational and epistemic approaches to protocol analysis: Bridging the gap. In *Proceedings of the 14th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'07)*, volume 4790 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag, Berlin, Germany, 2007.

Pierpaolo Degano and Corrado Priami. Proved trees. In Werner Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 629–640. Springer, 1992. ISBN 3-540-55719-9.

Wan Fokkink, Rob J. van Glabbeek, and Paulien de Wind. Compositionality of hennessy-milner logic by structural operational semantics. *Theoretical Computer Science*, 354(3):421–440, 2006.

Willem Jan (Wan) Fokkink and Thuy Duong Vu. Structural operational semantics and bounded nondeterminism. *Acta Informatica*, 39(6–7):501–516, 2003.

Willem Jan (Wan) Fokkink, Robert Jan (Rob) van Glabbeek, and Paulien de Wind. Compositionality of Hennessy-Milner logic through structural operational semantics. In Andrzej Lingas and Bengt J. Nilsson, editors, *Proceedings of the 14th Symposium on Fundamentals of Computation Theory (FCT'03)*, volume 2751 of *Lecture Notes in Computer Science*, pages 412–422. Springer-Verlag, Berlin, Germany, 2003.

Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, 2003. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/641909.641912.

Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Component verification with automatically generated assumptions. *Automated Software Engineering*, 12(3):297–320, 2005.

Jan Friso Groote. Transition system specifications with negative premises. *Theoretical Computer Science (TCS)*, 118(2):263–299, 1993.

Joseph Y. Halpern and Kevin R. O'Neill. Anonymity and information hiding in multiagent systems. *Journal of Computer Security*, 13(3):483–512, 2005.

Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Conference on Principles and Practice of Parallel Programming*, 2005.

Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and Jan van Leeuwen, editors, *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 1980. ISBN 3-540-10003-2.

Matthew Hennessy and Gordon D. Plotkin. Finite conjuncitve nondeterminism. In Klaus Voss, Hartmann J. Genrich, and Rozenberg Grzegorz, editors, *Concurrency and Nets, Advances in Petri Nets*, pages 233–244. Springer-Verlag, Berlin, Germany, 1987.

Matthew Hennessy and Colin Stirling. The power of the future perfect in program logics. *Information and Control*, 67(1-3):23–52, 1985.

Matthew C. B. Hennessy and A.J.R.G. (Robin) Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

Maurice Herlihy and J. Eliot B. Moss. Transactional support for lock free data structures. In *20th International Symposium on Computer Architecture*, June 1993.

Maurice Herlihy, Victor Luchango, and Mark Moir. A flexible framework for implementing software transactional memory. In *ACM SIGPLAN OOPSLA*, Oct 2006.

C.A.R. (Tony) Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

Arjen Hommersom, John-Jules Meyer, and Erik D. De Vink. Update semantics of security protocols. *Synthese*, 142(2):229+, 2004. ISSN 0039-7857. doi: 10.1007/s11229-004-2247-0. URL `http://dx.doi.org/10.1007/s11229-004-2247-0`.

Anna Ingólfsdóttir, Jens Christian Godskesen, and M. Zeeberg. Fra hennessy-milner logik til ccs-processer. Technical report, Aalborg Universitetscenter, 1987.

D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

Ruggero Lanotte and Simone Tini. Probabilistic congruence for semistochastic generative processes. In Vladimiro Sassone, editor, *Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'05)*, volume 3441 of *Lecture Notes in Computer Science*, pages 63–78. Springer-Verlag, 2005.

F. Laroussinie and Ph. Schnoebelen. Specification in CTL+Past for verification in CTL. *Information and Computation*, 156(1):236–263, 2000.

François Laroussinie and Kim Guldstrand Larsen. Compositional model checking of real time systems. In *CONCUR*, pages 27–41, 1995.

François Laroussinie and Kim Guldstrand Larsen. Cmc: A tool for compositional model-checking of real-time systems. In *FORTE*, pages 439–456, 1998.

Kim G. Larsen and Liu Xinxin. Compositionality through an operational semantics of contexts. *Journal of Logic and Computation*, 1(6):761–795, 1991.

A.J.R.G. (Robin) Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

A.J.R.G. (Robin) Milner. *Communication and Concurrency*. Prentice Hall, 1989.

MohammadReza Mousavi and Michel A. Reniers. Orthogonal extensions in structural operational. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1214–1225. Springer-Verlag, Berlin, Germany, 2005.

MohammadReza Mousavi, Michel Reniers, and Jan Friso Groote. A syntactic commutativity format for SOS. *Information Processing Letters (IPL)*, 93:217–223, March 2005.

MohammadReza Mousavi, Iain C.C. Phillips, Michel A. Reniers, and Irek Ulidowski. The meaning of ordered sos. In S. Arun-Kumar and N. Garg, editors, *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, volume 4337 of *Lecture Notes in Computer Science*, pages 334–345, Kolkata, India, 2006. Springer-Verlag.

MohammadReza Mousavi, Michel A. Reniers, and Jan Friso Groote. SOS formats and meta-theory: 20 years after. *Theoretical Computer Science*, (373):238–272, 2007.

Rocco De Nicola, Ugo Montanari, and Frits W. Vaandrager. Back and forth bisimulations. In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 1990. ISBN 3-540-53048-7.

Xavier Nicollin and Joseph Sifakis. The algebra of timed processes ATP: theory and application. *Information and Computation (I&C)*, 114(1):131–178, October 1994.

Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In M. Broy C. Hoare and R. Steinbrueggen, editors, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, pages 47–96. NATO ASI Series, IOS Press, 2001.

Simon Peyton Jones and Satnam Singh. A tutorial on parallel and concurrent programming in haskell. May 2008.

Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, 1996.

Iain C. C. Phillips and Irek Ulidowski. Reversing algebraic process calculi. In *FoSSaCS*, pages 246–260, 2006.

Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming (JLAP)*, 60–61:17–139, 2004a. This article first appeared as [Plotkin 1981].

Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming (JLAP)*, 60:3–15, 2004b.

Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, 31– 2 1977. IEEE Computer Society Press, Los Alamitos, CA, USA.

Franco Raimondi and Alessio Lomuscio. Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *Journal of Applied Logic*, 5(2):235 – 251, 2007. ISSN 1570-8683. doi: DOI:10.1016/j.jal.2005.12.010. URL `http://www.sciencedirect.com/science/article/B758H-4J2M4JD-2/2/e8c49b4a093f24ce79c4f235b1c63bc3`. Logic-Based Agent Verification.

Alex K. Simpson. Sequent calculi for process verification: Hennessy-milner logic for an arbitrary GSOS. *Journal of Logic and Algebraic Programming*, 60-61: 287–322, 2004.

Simone Tini. Rule formats for compositional non-interference properties. *Journal of Logic and Algebraic Progamming (JLAP)*, 60:353–400, 2004.

Irek Ulidowski and Shoji Yuen. Extending process languages with time. In Michael Johnson, editor, *Proceedings of the 6th International Conference Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 524–538. Springer-Verlag, Berlin, Germany, 1997.

Muck van Weerdenburg and Michel A. Reniers. Structural operational semantics with first-order logic. In Matthew Hennessy and Bartek Klin, editors, *Structural Operational Semantics (SOS'08), Preliminary Proceedings*, pages 48–62, 2008.

Chris Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic Journal of Computing*, 2(2):274–302, 1995.

W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2), Mar 2008.

Gaoyan Xie and Zhe Dang. Testing systems of concurrent black-boxes-an automata-theoretic and decompositional approach. In Wolfgang Grieskamp and Carsten Weise, editors, *Post-Proceedings of the 5th International Workshop on Formal Approaches to Software Testing (FATES'05)*, volume 3997 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2006.