Universal Temporal Concurrent Constraint Programming

Carlos Olarte

École Polytechnique
Thèse de Doctorat
Spécialité Informatique

Universal Temporal Concurrent Constraint Programming

Présentée et soutenue publiquement par

Carlos Olarte

le 29 septembre 2009

devant le jury composé de

Rapporteurs:	Ugo Montanari
	Thomas T. Hildebrandt
Directeur de thèse:	Catuscia Palamidessi
	Frank D. Valencia
Examinateurs:	Vijay A. Saraswat
	François Fages
	Marc Pouzet
	Moreno Falaschi

Contents

Acknowledgements

First of all, I want to express my gratitude for my supervisors Frank Valencia and Catuscia Palamidessi. Their dedication and enthusiasm for Computer Sciences have inspired me during the past three years. They have always guided me in the right direction and provided a warm environment to grow as a researcher and as a person.

I owe much to Camilo Rueda for giving me constant advice and support during my PhD. I have to say that the outstanding work of Camilo motivated me to pursue an academic career. I am also indebted to Moreno Falaschi with whom I had the pleasure to work during these three years.

Thanks to INRIA and Javeriana University to fund my PhD and to the Laboratoire d'Informatique de l'École Polytechnique for allowing me to develop my thesis there.

My gratitude also goes to Ugo Montanari and Thomas Hildebrandt for taking some of their precious time to review this document. I am also grateful to Vijay Saraswat , François Fages, Moreno Falaschi and Marc Pouzet to be part of my jury.

I want also to express my gratitude to the members of the AVISPA research group, specially to Hugo López for working on the `utcc` calculus and Jorge A. Pérez for his criticisms. Many thanks to Camilo Rueda, Jesús Aranda, Jorge Pérez and Michael Martínez to proofread some of the chapters of these dissertation.

Special thanks to my colleagues at LIX, Romain Beauxis, Sylvain Pradalier, Florian Richoux, Jesus Aranda, Peng Wu, Angelo Troina and Ulrich Herberg. My gratitude also goes to the Brazilian community at LIX, Elaine Pimentel, Vivek Nigam and Mario Sergio Alvim. Thanks to you I have a better understanding of the word "saudade".

I also want to show my affection for my friends Liliana Rosero, Alejandro Arbelaez, Andrea Hacker, Julianne Monceaux and Anna Johnsson. Many thanks to my friends in Colombia Carolina Jimenez, Diego Polo and Carlos Restrepo, to support me during these three years of being far from home.

I want to say also that I am happy to be a member of the Valencia-Södergren family. Sara, Frank and Felipe welcomed me from the first moment we met.

Many thanks to the administrative staff at the École Polytechnique that kindly helped me with the bureaucratic process of being a foreigner in France. Special thanks to Lydie Fontaine, Audrey Lemarechal and Isabelle Biercewicz.

I would also like to express how fortunate I feel to have spent three years of my life in the beautiful city of Paris. It was so exciting culturally and gastronomically speaking :). Many museums, exciting places and cultural activities that changed in some way my life. I will always keep in my heart the experiences of having dinner in "La Truffière", "Vin et Maré", "Pain, Vin, Fromages" and "Pot de Terre" among several others. Thanks to Elaine, Catuscia and Moreno to share my passion for the french and italian cuisines and wines.

I warmly thank to my wife Carolina Ramírez who found the way to make these three years abroad more bearable. I thank from the bottom of my heart my parents Sonia Vega and Carlos Olarte. They always encouraged me and showed me their affection. Many thanks also to my brother Andrés Olarte and his wife Alexandra Franco to take the risk of trying my recipes. Finally, thanks to Gloria Olarte, Cesar Augusto Ramirez and Paula Ramírez to help Carolina and me during these difficult years.

Carlos A. Olarte,
June the 14th, 2009.

Abstract

Concurrent Constraint Programming (CCP) [Saraswat 1993] is a formalism for concurrency in which agents (processes) interact with one another by telling (adding) and asking (reading) information represented as constraints in a shared medium (the store). Temporal Concurrent Constraint Programming (`tcc`) extends CCP by allowing agents to be constrained by time conditions. This dissertation studies temporal CCP as a model of concurrency for mobile, timed reactive systems. The study is conducted by developing a process calculus called `utcc`, Universal Temporal CCP. The thesis is that `utcc` is a model for concurrency where behavioral and declarative reasoning techniques coexist coherently, thus allowing for the specification and verification of mobile reactive systems in emergent application areas.

The `utcc` calculus generalizes `tcc` [Saraswat 1994], a temporal CCP model of reactive synchronous programming, with the ability to express mobility. Here mobility is understood as communication of private names as typically done for mobile systems and security protocols. The `utcc` calculus introduces parametric ask operations called abstractions that behave as persistent parametric asks during a time-interval but may disappear afterwards. The applicability of the calculus is shown in several domains of Computer Science. Namely, decidability of Pnueli's First-order Temporal Logic, closure-operator semantic characterization of security protocols, semantics of a Service-Oriented Computing language, and modeling of Dynamic Multimedia-Interaction systems.

The `utcc` calculus is endowed with an operational semantics and then with a symbolic semantics to deal with problematic operational aspects involving infinitely many substitutions and divergent internal computations. The novelty of the symbolic semantics is to use temporal constraints to represent finitely infinitely-many substitutions.

In the tradition of CCP-based languages, `utcc` is a declarative model for concurrency. It is shown that `utcc` processes can be seen, at the same time, as computing agents and as logic formulae in the Pnueli's First-order Linear-time Temporal Logic (FLTL) [Manna 1991]. More precisely, the outputs of a process correspond to the formulae entailed by its FLTL representation.

The above-mentioned FLTL characterization is here used to prove an insightful decidability result for Monadic FLTL. To do this, it is proven that in contrast to `tcc`, `utcc` is Turing-powerful by encoding Minsky machines [Minsky 1967]. The encoding uses a simple decidable constraint system involving only monadic predicates and no equality nor function symbols. The importance of using such a constraint system is that it allows for using the underlying theory of `utcc` to prove the undecidability of the validity problem for monadic FLTL without function symbols nor equality. In fact, it is shown that this fragment of FLTL is incomplete (its set of tautologies is not recursively enumerable). This result refutes a decidability conjecture for FLTL from a previous work. It also justifies the restriction imposed in previous decidability results on the quantification of flexible-variables. This dissertation then fills a gap on the decidability study of monadic FLTL.

Similarly to `tcc`, `utcc` processes can be semantically characterized as partial closure operators. Because of the additional technical difficulties posed by `utcc`, the codomain of the closure operators is more involved than that for `tcc`. Namely, processes are mapped into sequences of future-free temporal formulae rather than sequences of basic constraints as in `tcc`. This representation is shown to be fully abstract with respect to the input-output behavior of processes for a meaningful fragment of the calculus. This shows that mobility can be captured as closure operators over an underlying constraint system.

As a compelling application of the semantic study of `utcc`, this dissertation gives a

closure operator semantics to a language for security protocols. This language arises as a specialization of `utcc` with a particular cryptographic constraint systems. This brings new semantic insights into the modeling and verification of security protocols.

The `utcc` calculus is also used in this dissertation to give an alternative interpretation of the π-based language defined by Honda, Vasconcelos and Kubo (`HVK`) for structuring communications [Honda 1998]. The encoding of `HVK` into `utcc` is straightforwardly extended to explicitly model information on session duration, allows for declarative preconditions within session establishment constructs, and features a construct for session abortion. Then, a richer language for the analysis of sessions is defined where time can be explicitly modeled. Additionally, relying on the above-mentioned interpretation of `utcc` processes as FLTL formulae, reachability analysis of sessions can be characterized as FLTL entailment.

It is also illustrated that the `utcc` calculus allows for the modeling of dynamic multimedia interaction systems. The notion of constraints as partial information neatly defines temporal relations between interactive agents or events. Furthermore, mobility in `utcc` allows for the specification of more flexible and expressive systems in this setting, thus broadening the interaction mechanisms available in previous models.

Finally, this dissertation proposes a general semantic framework for the data flow analysis of `utcc` and `tcc` programs by abstract interpretation techniques [Cousot 1979]. The concrete and abstract semantics are compositional reducing the complexity of data flow analyses. Furthermore, the abstract semantics is parametric with respect to the abstract domain and allows for reusing the most popular abstract domains previously defined for logic programming. Particularly, a groundness analysis is developed and used in the verification of a simple reactive systems. The abstract semantics allows also to efficiently exhibit a secrecy flaw in a security protocol modeled in `utcc`.

Keywords: Concurrent Constraint Based Calculi, Denotational Semantics, Symbolic Semantics, Security Protocols, First-Order Linear-Time Temporal Logic.

Chapter 1

Introduction

Nowadays concurrent and *mobile* systems are ubiquitous in several domains and applications. They pervade different areas in science (e.g. biological and chemical systems), engineering (e.g., security protocols and mobile and service oriented computing) and even the arts (e.g. tools for multimedia interaction).

In general, concurrent systems exhibit complex forms of interaction, not only among their internal components, but also with the surrounding environment. When mobility is also considered, an additional burden one has to deal with is the fact that the internal configuration and communication structure of the system evolve while interacting.

A legitimate challenge is then to provide computational models allowing to understand the nature and the behavior of such complex systems as the observation of the evolution and interaction of their components.

As an answer to this challenge, process calculi such as CCS [Milner 1989], the π-calculus [Milner 1999, Sangiorgi 2001] and CSP [Hoare 1985] among several others have arisen as mathematical formalisms to model and reason about concurrent systems. They treat concurrent processes much like the λ-calculus treats computable functions. They then provide a language in which the structure of terms represents the structure of processes together with an operational semantics to represent computational steps.

Arguably, the π-calculus [Milner 1999, Milner 1992b, Sangiorgi 2001] is one of the most notable and simple computational models to describe concurrent and *mobile* systems. In this model, mobility is understood as generation and communication of private names or links. Roughly speaking, processes in the π-calculus *interact* by creating, and *synchronously* sending and receiving communications names (or links). Thus, one can see that the processes evolve changing their communication structure during computation.

As an alternative to models for concurrency based on the π-calculus, Concurrent Constraint Programming (CCP) [Saraswat 1993] has emerged as a model for concurrency that combines the traditional operational view of process calculi with a *declarative* one based upon logic. This combination allows CCP to benefit from the large body of reasoning techniques of both process calculi and logic. In fact, CCP has successfully been used in the modelling and verification of several concurrent scenarios: E.g., timed, reactive and stochastic systems. See e.g., [Saraswat 1993, Saraswat 1994, Nielsen 2002a, Buscemi 2007, Gupta 1996a, Hildebrandt 2009, Rueda 2004, Olarte 2008a, Bortolussi 2008].

Agents in CCP *interact* with each other by telling and asking information represented as constraints (e.g, $x > 42$) in a global store. The type of constraints (i.e. basic constructs) is not fixed but *parametric* in an underlying constraint system defining the vocabulary of assertions the processes can use [Saraswat 1993] .

The basic constructs in CCP are the process **tell**(c) adding the constraint c to the store, thus making it available to the other processes; and the *positive ask* **when** c **do** P querying if the current store is strong enough to entail the guard c; if so, it behaves like P. Otherwise it remains blocked until more information is added to entail the constraint c. This way, ask processes define a synchronization mechanism based on entailment of constraints.

Apart from the above mentioned operations, and similarly to most process calculi, CCP languages feature constructs for information hiding and parallel composition: The process

(**local** x) P declares a private variable x for P. The process $P \parallel Q$ stands for the parallel execution of P and Q possibly communicating through the shared store.

Interaction of CCP processes is then *asynchronous* as communication takes place through the shared store of partial information. Similar to other formalisms, by defining local (or private) variables, CCP processes specify boundaries in the interface they offer to interact with each other. Once these interfaces are established, there are few mechanisms to modify them. This is not the case e.g., in the π-calculus where processes can change their communication patterns by exchanging their private names.

In this dissertation, we aim at developing a theory for a CCP-based model where processes may change their interaction mechanisms by communicating their local names, i.e., they can exhibit *mobile* behavior in the sense of the π-calculus.

The novelty of the model we propose relies on two design criteria that distinguish CCP-based calculi from other formalisms:

(1) *Logic correspondence*, which provides CCP with a unique declarative view of processes: processes can be seen, at the same time, as computing agents and logic formulae. This allows for example for rechability analysis using deduction in logic [Saraswat 1994, de Boer 1997, Nielsen 2002a].

(2) *Determinism*, which is the source of CCP's elegant and simple semantic characterizations. For example, semantics based on closure operators [Scott 1982], i.e. extensive, monotonic and idempotent functions, as in [Saraswat 1993, Saraswat 1991, Saraswat 1994].

This thesis then strives for finding a concurrency model for the specification of mobile reactive system where *logic* and *behavioral* approaches coexist coherently. Doing that, we bring new reasoning techniques for the modeling and verification of systems in emergent application areas as we describe below.

Before describing our approach, let us discuss previous studies of mobility in the context of CCP. Basic CCP is able to specify mobile behavior using logical variables to represent channels and unification to bind messages to channels [Saraswat 1993]. In this approach, if two messages are sent through the same channel, they must be equal. In other case an inconsistency arises. In [Laneve 1992] this problem is solved by using *Atomic CCP* where **tell**(c) adds c to the current store d if $c \wedge d$ is not inconsistent. Here a protocol is required since messages must compete for a position in a list representing the messages previously sent. Atomic tells then introduce non-determinism to the calculus since executing or not **tell**(c) depends on the current store.

The approach in [Buscemi 2007, Buscemi 2008] combines the CCP model with the name-passing discipline of the π-F calculus [Wischik 2005] leading to the cc-pi calculus. In this setting, CCP processes are allowed to send and receive communication channels by means of the constructs inherited from π-F. Nevertheless, the cc-pi calculus is not deterministic and does not feature a declarative view of processes as formulae in logic.

Mobility can be also modelled in CCP by adding *linear* parametric ask processes as done in Linear CCP [Fages 2001, Saraswat 1992]. A parametric ask $A(x)$ can be viewed as a process **when** c **do** P with a variable x declared as a formal parameter. Intuitively, $A(x)$ may evolve into $P[y/x]$, i.e. P with x replaced by y, if $c[y/x]$ is entailed by the store. Mobility is exhibited when y is a private variable (*link*) from some other process. This extension, however, is *non-deterministic:* If both $c[y/x]$ and $c[z/x]$ are entailed by the store, $A(x)$ may evolve to either $P[y/x]$ or $P[z/x]$.

The above kind of non-determinism can be avoided by extending CCP only with *persistent* parametric asks following the semantics of the persistent π-calculus in [Palamidessi 2006]. The idea is that if both $c[y/x]$ and $c[z/x]$ are entailed by the store, a persistent $A(x)$ evolves into $A(x) \parallel P[y/x] \parallel P[z/x]$. Forcing every ask to be persistent, however, makes the extension not suitable for modelling typical scenarios where a process stops after performing its query (e.g., web-services requests).

Our approach is to extend CCP with *temporary* parametric ask operations. Intuitively, these operations behave as persistent parametric asks during a *time-interval* but may disappear afterwards. We do this by generalizing the timed CCP model in [Saraswat 1994]. We call this extension *Universal Timed CCP* (`utcc`).

In `utcc`, like in `tcc`, time is conceptually divided into *time intervals* (or *time units*). In a particular time interval, a CCP process P gets an input c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines a residual process Q which is then executed in the next time interval. The resulting store d *is not automatically transferred* to the next time interval. This view of reactive computation is particularly appropriate for programming reactive systems in the sense of Synchronous Languages [Berry 1992], i.e., systems that react continuously with the environment at a rate controlled by the environment such as controllers or signal-processing systems.

The fundamental move in `utcc` is to replace the `tcc` ask operation **when** c **do** P with a *temporary parametric ask* of the form $(\mathbf{abs}\ \vec{x}; c)\, P$. This process can be viewed as a *lambda abstraction* of the process P on the variables $\vec{x}$ under the constraint (or with the *guard*) c. Intuitively, $Q = (\mathbf{abs}\ \vec{x}; c)\, P$ executes $P[\vec{t}/\vec{x}]$ in the current time interval for *all the sequences of terms* $\vec{t}$ s.t $c[\vec{t}/\vec{x}]$ is entailed by the current store. Furthermore, Q evolves into **skip** (representing inaction) after the end of the time unit, i.e. abstractions are not persistent when passing from one time unit to the next one.

We shall show that the new construct in `utcc` has a pleasant duality with the local operator: From a programming language perspective, $\vec{x}$ in $(\mathbf{local}\, \vec{x}; c)\, P$ can be viewed as the local variables of P while $\vec{x}$ in $(\mathbf{abs}\ \vec{x}; c)\, P$ can be viewed as the formal parameters of P. From a logical perspective, these processes correspond, respectively, to the *existential* and the *universal* formulae $\exists \vec{x}(c \wedge F_P)$ and $\forall \vec{x}(c \Rightarrow F_P)$ where F_P corresponds to P.

In this dissertation we shall also show that the interplay of the local and the abstraction operator in `utcc` allows for communication of private names, i.e., mobility as is understood in the π-calculus. This way, we provide a model for concurrency to specify *mobile reactive systems* that complies with the criteria (1) and (2) above. More precisely, we shall show a strong correspondence between `utcc` processes and formulae in first-order linear-time temporal logic [Manna 1991]. Furthermore, `utcc` being deterministic, allows for an elegant semantic characterization of processes as closure operators. We shall show that this allows us to capture compositionally the behavior of processes.

We shall also show that the `utcc` calculus has interesting applications in meaningful concurrent scenarios in several domains of Computer Science. Namely, decidability of Pnueli's First-order Temporal Logic [Manna 1991], closure-operator semantic characterization of security protocols, semantics of a Service-Oriented Computing languages, and modeling of Dynamic Multimedia-Interaction systems. We elaborate more on these results next.

1.1 Contributions and Organization

In what follows we describe the structure of this dissertation and its contributions. Each chapter concludes with a summary of its content and a discussion about related work. Frequently used notational conventions and terminology are summarized in the Index.

Chapter 2 [Background]. In this chapter we introduce the basic concepts and terminology used throughout this dissertation. We briefly describe the Concurrent Constraint Programming model and the ideas from process calculi, reactive systems and temporal logics that motivated the development of `utcc`.

Chapter 3 [Operational Semantics]. In the same lines of `tcc` [Saraswat 1994, Nielsen 2002a], we give `utcc` an *operational semantics* defined by an internal and an observable transition relation. The first one describes the evolution of processes during a time unit. The second one describes how, given an input from the environment, a process reacts outputting the final store obtained from a finite number of internal reductions. This way, we define the input-output behavior of processes. Finally, we show that `utcc` allows for mobility and it is deterministic.

Chapter 4 [Symbolic Semantics]. Due to the abstraction operator in `utcc`, some processes may exhibit infinitely many internal reductions when considering the operational semantics of Chapter 3. We solve this problem by endowing `utcc` with a novel *symbolic semantics* that uses temporal constraints to represent finitely a possible infinite number of substitutions. This way, we can observe the behavior of processes exhibiting infinitely many internal reductions. For instance, those arising in the verification of security protocols where the model of the attacker may generate an unbound number of messages (constraints). To our knowledge, this is the first symbolic semantics in concurrency theory using temporal constraints as finite representations of substitutions.

Chapter 5 [Logic Characterization]. In addition to the usual behavioral techniques from process calculi, CCP enjoys a declarative view of processes based upon logic. This makes CCP a language suitable for both the specification and implementation of programs. In this chapter, we show that the `utcc` calculus is a declarative model for concurrency. We do this by exhibiting a strong correspondence of `utcc` and First-Order Linear-Time Temporal Logic (FLTL) [Manna 1991]. This way, processes can be seen, at the same time, as computing agents and FLTL formulae. The logic characterization we propose allows for using well-established techniques from FLTL for reachability analysis of `utcc` processes. For instance, we can show if there is a way to reach a state in a security protocol where an intruder knows a secret, i.e., there is a secrecy breach.

Chapter 6 [Expressiveness and Decidability of FLTL]. The computational *expressiveness* of `tcc` languages have been thoroughly studied in the literature allowing for a better understanding of `tcc` and its relation with other formalisms. In particular, [Saraswat 1994] and [Valencia 2005] shows that `tcc` processes can be represented as *finite-state* Büchi automata [Buchi 1962] and thus cannot encode Turing-powerful formalisms. In this chapter we show the full computational expressiveness of `utcc` and its compositional correspondence to functional programming: we provide an encoding of *Minsky machines* and the λ-*calculus* into well-terminated `utcc` processes, i.e. processes that do not exhibit infinite internal computations. Although both formalisms are Turing-equivalent these encodings serve two different purposes. On the

one hand, the encoding of Minsky machines uses a very simple constraint system: the monadic fragment without equality nor function symbols of first-order logic. On the other hand, the encoding of the λ-calculus uses instead a *polyadic* constraint system but it is *compositional* unlike that of Minsky machines. This encoding is a significant application showing how `utcc` is able to mimic one of the most notable and simple computational models achieving Turing completeness.

As an application of this expressiveness study, we use the FLTL characterization in Chapter 5 and the encoding of Minsky machines to prove an insightful (un)decidability result for Monadic FLTL. We prove the monadic fragment of FLTL without equality nor function symbols to be strongly incomplete, and then, undecidable its validity problem. This result clarifies previous decidability results and conjectures in the literature. This dissertation then fills a gap on the decidability study of monadic FLTL.

Chapter 7 [Denotational Semantics]. By building on the semantics of `tcc` in [Saraswat 1994, Nielsen 2002a], we show that the input-output behavior of `utcc` processes can be characterized as a closure operator. Because of additional technical difficulties posed by `utcc`, the codomain of the closure operators is more involved than that for `tcc`. Namely, we shall use sequences of future-free temporal formulae (constraints) rather than sequences of basic constraints as in `tcc`. Next, we give a compositional denotational account of this closure-operator characterization. We show that the denotational model is fully abstract with respect to the symbolic input-output behavior of processes for a significant fragment of the calculus. This in particular shows that mobility in `utcc` can be elegantly represented as closure operators over some underlying constraint system.

Chapter 8 [Closure Operators for Security]. As a compelling application of the denotational account of `utcc`, we shall bring new semantic insights into the modeling of security protocols. We identify a process language for security protocols that can be represented as closure operators. This language arises as a parameterization of `utcc` with a particular cryptographic constraint systems. We shall argue that the interpretation of the behavior of protocols as closure operators is a natural one. For instance, a spy can only produce new information (extensiveness); the more information she gets, the more she will infer (monotonicity); and she infers as much as possible for the information she gets (idempotence). To our knowledge no closure operator denotational account has previously been given in the context of calculi for security protocols.

Chapter 9 [Other Applications]. The `utcc` calculus was not specifically designed for the modeling and verification of security protocols but to model in general mobile reactive systems. In this chapter we show that `utcc` has much to offer in the specification and verification of systems in two emergent application areas.

- **Service Oriented Computing.** We give an alternative interpretation of the π-based language defined by Honda, Vasconcelos and Kubo (`HVK`) for structuring communications [Honda 1998]. The encoding of `HVK` into `utcc` is straightforwardly extended to provide a richer language for the analysis of sessions where time can be explicitly modeled. Relying on the FLTL characterization of `utcc`, we show that it is possible to perform reachability analysis of sessions.
- **Multimedia Interaction Systems.** As second application domain, we shall illustrate that the `utcc` calculus allows for the modeling of dynamic multime-

dia interaction systems. The notion of constraints as partial information neatly defines temporal relations between interactive agents or events. Furthermore, mobility in `utcc` allows for the specification of more flexible and expressive systems in this setting, thus broadening the interaction mechanisms available in previous models.

Chapter 10 [Static Analysis]. In this chapter we propose a semantic framework for the static analysis of `utcc` and `tcc` programs based on abstract interpretation techniques [Cousot 1977]. The abstract semantics proposed is compositional, thus allowing us to reduce the complexity of data flow analyses. Furthermore, it effectively approximates the behavior of `utcc` programs. The proposed framework is parametric with respect to the abstract domain and then, different analyses can be performed by instantiating it. We illustrate how it is possible to reuse abstract domains previously defined for logic programming to perform, e.g., a groundness analysis of a `tcc` program. Furthermore, we make also use of the abstract semantics to *automatically* exhibit a secrecy flaw in a security protocol.

Chapter 11 [Concluding Remarks]. This chapter presents an overview of this dissertation and gives some directions for future work.

1.2 Publications from this Dissertation

Most of the material of this dissertation has been previously reported in the following works.

- **Proceedings of conferences.**
  - C. Olarte and F. Valencia. *The Expressivity of Universal Timed CCP: Undecidability of Monadic FLTL and Closure Operators for Security.* In Proc of PPDP'08: 8-19. ACM Press. 2008 [Olarte 2008b].

    The main contributions of this paper are included in Chapters 5, 6, 7 and 8.
  - Carlos Olarte and Frank D. Valencia. *Universal Concurrent Constraint Programing: Symbolic Semantics and Applications to Security.* In Proc. of SAC 2008. ACM Press, 2008.

    The main contributions of this paper are included in Chapters 3, 4 and 5.
  - M. Falaschi, C. Olarte and C. Palamidessi. *A Framework for Abstract Interpretation of Timed Concurrent Constraint Programs.* In Proc of PPDP09. ACM Press. 2009 (To appear) [Falaschi 2009].

    The main contributions of this paper are included in Chapter 10.
  - M. Falaschi, C. Olarte, C. Palamidessi and F. Valencia. *Declarative Diagnosis of Temporal Concurrent Constraint Programs.* In Proc. of ICLP 2007: 271-285. Springer. 2007 [Falaschi 2007].

    The main contributions of this paper are included in Chapter 10.
  - Carlos Olarte and Camilo Rueda. *A declarative language for dynamic multimedia interaction systems.* In Proc of. MCM'09 (to appear). Springer, 2009 [Olarte 2009b].

    The main contributions of this paper are included in Chapter 9.

- **Proceedings of workshops.**
  - H. A. López, C. Olarte, and J. A. Pérez. *Towards a Unified Framework for Declarative Structured Communications.* In Proc. of PLACES'09, February 2009 [Lopez 2009].

    The main contributions of this paper are included in Chapter 9.
- **Book Chapters.**
  - C. Olarte, C. Rueda and F. Valencia. *Concurrent Constraint Calculi: a Declarative Paradigm for Modeling Music Systems.* Nouveaux paradigmes pour l'informatique musicale: 93-112. Delatour France / IRCAM-Centre Pompidou. 2009 [Olarte 2009a].
- **Abstracts and Short Papers.**
  - C. Olarte, C. Palamidessi and F. Valencia. *Universal Timed Concurrent Constraint Programming* (Abstract). ICLP 2007: 464-465. Springer-Verlag. 2007 [Olarte 2007b].
  - Jesús Aranda, Gerard Assayag, Carlos Olarte, Jorge A. Pérez, Camilo Rueda, Mauricio Toro and Frank D. Valencia. *An Overview of FORCES: An INRIA Project on Declarative Formalisms for Emergent Systems.* To appear in Proc. of ICLP 2009 [Aranda 2007].
- **Others**
  - C. Olarte. *A Process Calculus for Universal Concurrent Constraint Programming: Semantics, Logic and Application.* Vol. 20 n. 3/4, December 2007 of the Association for Logic Programming (ALP) Newsletter [Olarte 2007a].
  - C. Olarte, C. Rueda and Frank D. Valencia. *Concurrent Constraint Programming: Calculi, Languages and Emerging Applications.* Vol. 21 n. 2-3, August 2008 of the Association for Logic Programming (ALP) Newsletter [Olarte 2008a].

Chapter 2

Preliminaries

In this chapter we introduce the basic concepts and terminology used throughout this dissertation. We briefly describe the Concurrent Constraint Programming model and the ideas from process calculi, reactive systems and temporal logics that motivated the development of `utcc`. We do not intent to give an in-depth review of these concepts but rather to contextualize the development of `utcc` in this thesis. We encourage the reader to follow the references to have a complete description of each topic addressed in this chapter.

2.1 Process Calculi

Process calculi such as CCS [Milner 1989] , CSP [Hoare 1985], the process algebra ACP [Bergstra 1985, Baeten 1990] and the π-calculus [Milner 1999, Sangiorgi 2001] are among the most influential formal methods for reasoning about concurrent systems. A common feature of these calculi is that they treat processes much like the λ-calculus treats computable functions. For example, a typical process term is the parallel composition $P \parallel Q$, which is built from the terms P and Q with the constructor $\parallel$ and represents the process that results from the parallel execution of the processes P and Q. An operational semantics may dictate that if P can reduce to (or evolve into) P', written $P \longrightarrow P'$, then we can also have the reduction $P \mid Q \longrightarrow P' \mid Q$.

Process calculi in the literature mainly agree in their emphasis upon algebra. The distinctions among them arise from issues such as the process constructions considered (i.e., the language of processes), the methods used for giving meaning to process terms (i.e. the semantics), and the methods to reason about process behavior (e.g., process equivalences or process logics). Some other issues addressed in the theory of these calculi are their expressive power, and analysis of their behavioral equivalences.

The `utcc` process calculus aims at modeling *mobile reactive systems*, i.e., systems that interact continuously with the environment and may change their communication structure. In this dissertation mobility is understood as generation and communication of private links or channels much like in the π-calculus [Milner 1999, Sangiorgi 2001], one of the main representative formalisms for mobility in concurrency theory. Mobility is fundamental, e.g., to specify security protocols where *nonces* (i.e, randomly-generated unguessable items) are transmitted. In the next section we give a brief introduction of the π-calculus. This will help us to better understand the notion of mobility we address in this dissertation and also the applications we describe in Chapters 8 and 9.

2.2 The π-calculus

The π-calculus [Milner 1999, Milner 1992b, Sangiorgi 2001] is a process calculus aiming at describing *mobile* systems whose configuration may change during the computation. Similar to the λ-calculus, the π-calculus is minimal in that it does not contain primitives such as numbers, booleans, data structures, variables, functions, or even the usual flow control statements.

Mobility in the π-calculus. As we said before, mobility in the π-calculus is understood as generation and communication of private names or links. Links between processes can be created and communicated, thus changing the communication structure of the system. This also allows us to consider the location of an agent in an interactive system to be determined by the links it possesses, i.e. which other agents it possesses as neighbors.

2.2.1 Names and Actions

Names are the most primitive entities in the π-calculus. The simplicity of the calculus relays on the dual role that they can play as communication channels and variables.

We presuppose a countable set of (ports, links or channels) names, ranged over by $x, y, ...$ For each name x, we assume a co-name $\overline{x}$ thought of as complementary, so that $x = \overline{\overline{x}}$. We shall use $l, l', ...$ to range over names and co-names.

Definition 2.2.1 (π-calculus syntax)**.** *Process in the π-calculus are built from names by the following syntax:*

$$\begin{array}{lcl} P, Q, ... & := & \sum_{i \in I} (\alpha_i.P_i) \mid (\nu x)P \mid P \mid Q \\ \alpha & := & \overline{x}y \mid x(y) \end{array}$$

where I is a finite set of indexes.

We shall recall briefly some notions as well as the intuitive behavior of the constructs above. See [Milner 1999, Milner 1992b, Sangiorgi 2001] for further details.

The construct $\sum_{i \in I} \alpha_i.P_i$ represents a process able to perform one -but only one- of its α_i's actions and then behave as the corresponding P_i . When $|I| = 0$ we shall simply write 0 (i.e. the inactive process). The actions prefixing the Pi's can be of two forms: An *output* $\overline{x}y$ and an *input* $x(y)$. In both cases x is called the subject and y the object. The action $\overline{x}y$ represents the capability of sending the name y on channel x. The action $x(y)$ represents the capability of receiving the name, say z, on channel x and replacing y with z in its corresponding continuation. Furthermore, in $x(y).P$ the input action binds the name y in P. The other name binder is the restriction $(\nu x)P$ that declares a name x private to P, hence bound in P. Given a process Q, we define in the standard way its bound names $bn(Q)$ as the set of variables with a bound occurrence in Q, and its free names $fn(Q)$ as the set of variables with a non-bound occurrence in Q.

Finally, the process $P \mid Q$ denotes parallel composition ; P and Q running in parallel.

2.2.2 Operational Semantics

The above intuition about process behavior in the π-calculus is made precise by the rules in Table 2.1. The reduction relation $\longrightarrow$ is the least binary relation on processes satisfying the rules in Table 2.1. These rules are easily seen to realize the above intuition. We shall use $\longrightarrow^*$ to denote the reflexive and transitive closure of $\longrightarrow$. A reduction $P \longrightarrow Q$ basically says that P can evolve, after some communication between its subprocesses, into Q. The reductions are quotiented by the structural congruence relation $\equiv$ which postulates some basic process equivalences.

Definition 2.2.2 (π-calculus Structural Congruence)**.** *Let $\equiv$ be the smallest congruence over processes satisfying the following axioms:*

1. *$P \equiv Q$ if P and Q differ only by a change of bound names (α-conversion).*

$$\text{REACT}\ \frac{}{(x(y).P + M) \mid (\overline{x}(z).Q + N) \longrightarrow P[z/y] \mid Q}$$

$$\text{PAR}\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \text{RES}\ \frac{P \longrightarrow P'}{(\nu a)(P) \longrightarrow (\nu a)P'}$$

$$\text{STRUCT}\ \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

Table 2.1: Reductions Rules for the π-calculus.

2. $P \mid 0 \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$.

3. *If* $x \notin fn(P)$ *then* $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$.

4. $(\nu x)0 \equiv 0$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$.

2.2.3 An Example of Mobility

Let us extend the syntax in Definition 2.2.1 with the *match* process $[x = y].P$. This process behaves as P if x and y are the same name, otherwise it does nothing.

Consider now the following processes:

$$\begin{array}{rcl} P & = & c_1(y).\overline{c_2}y \\ Q & = & (\nu z)(\overline{c_1}z.c_2(z').[z = z'].R) \end{array}$$

Intuitively, if a link y is sent on channel c_1, P forwards it on channel c_2. Now, Q sends its private link z on c_1 and if it gets it back on c_2 it executes the process R.

Using the rules in Table 2.1 we can verify the following

$$\begin{array}{rcl} P \mid Q & \longrightarrow^* & (\nu z)(c_1(y).\overline{c_2}y \mid \overline{c_1}z.c_2(z').[z = z'].R) \\ & \longrightarrow^* & (\nu z)(\overline{c_2}z \mid c_2(z').[z = z'].R) \\ & \longrightarrow^* & (\nu z)(([z = z'].R)[z/z']) \\ & \longrightarrow^* & (\nu z)(R[z/z']) \end{array}$$

This means that the parallel composition of P and Q leads to a configuration where the process R is executed. We shall come back to this example in Section 3.6 where we show that the `utcc` calculus is able to mimic the mobile behavior of the processes above, where the private name z of Q is sent to P.

2.3 Concurrent Constraint Programming

Concurrent Constraint Programming (CCP) [Saraswat 1993, Saraswat 1991] has emerged as a simple but powerful paradigm for concurrency tied to logic. CCP extends and subsumes both concurrent logic programming [Shapiro 1989] and constraint logic programming [Jaffar 1987]. A fundamental feature in CCP is the specification of concurrent systems by means of constraints. A constraint (e.g. $x + y \geq 10$) represents partial information about certain variables. During the computation, the current state of the system is specified by a set of constraints called the *store*. Processes can change the state of the system by telling

information to the store (i.e., adding constraints), and synchronize by asking information to the store (i.e., determining whether a given constraint can be inferred from the store).

Like done in process calculi, the language of processes in the CCP model is given by a small number of primitive operators or combinators. A typical CCP process language features the following operators:

- A *tell* operator adding a constraint to the store.
- An *ask* operator querying if a constraint can be deduced from the store.
- *Parallel Composition* combining processes concurrently.
- A *hiding* operator (also called *restriction* or *locality*) introducing local variables and thus restricting the interface a process can use to interact with others.

2.3.1 Reactive Systems and Timed CCP

Reactive systems [Berry 1992] are those that react continuously with their environment at a rate controlled by the environment. For example, a controller or a signal-processing system, receive a stimulus (input) from the environment. It computes an output and then, waits for the next interaction with the environment.

Languages such as Esterel [Berry 1992], Lustre [Halbwachs 1991], Lucid Synchrone [Caspi 1999] and Signal [Benveniste 1991] among others have been proposed in the literature for programming reactive systems. Those languages are based on the hypothesis of *Perfect Synchrony*: Program combinators are determinate primitives that respond instantaneously to input signals.

The timed CCP calculus (`tcc`) [Saraswat 1994] extends CCP for reactive systems. The fundamental move in the `tcc` model is then to extend the standard CCP with delay and time-out operations. The delay operation forces the execution of a process to be postponed to the next time interval. The time-out operation waits during the current time interval for a given piece of information to be present and if it is not, triggers a process in the next time interval.

Time in `tcc` is conceptually divided into *time intervals* (or *time units*). In a particular time interval, a CCP process P gets as input a constraint c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines also a residual process Q which is then executed in the next time unit. The resulting store d is not automatically transferred to the next time unit. This way, computations during a time unit proceed monotonically but outputs of two different time units are not supposed to be related to each other.

We postpone the presentation of the syntax and the operational semantics of CCP and `tcc` to Chapter 3 where we introduce the `utcc` calculus.

2.4 First-Order Linear-Time Temporal Logic

Temporal logics were introduced into computer science by Pnueli [Pnueli 1977] and thereafter proven to be a good basis for specification as well as for (automatic and machine-assisted) reasoning about concurrent systems.

In this dissertation, we shall show that `utcc` is a *declarative* model for concurrency. More precisely, we shall show that `utcc` processes can be seen, at the same time, as computing agents and formulae in First-Order Linear-Time Temporal Logic (FLTL) [Manna 1991].

Furthermore, the symbolic semantics that we develop in Chapter 4 makes use of temporal formulae to give a finite representation of a possible infinite number of substitutions in the operational semantics.

For those reasons, in this section we recall the syntax and semantics of FLTL. We refer the reader to [Manna 1991] for further details.

Recall that a signature Σ is a set of constant, function and predicate symbols. A first-order language $\mathcal{L}$ is built from the symbols in Σ, a denumerable set of variables $x, y, \ldots$, and the logic symbols $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall$, `true` and `false`.

Definition 2.4.1 (FLTL Syntax)**.** *Given a first-order language $\mathcal{L}$, the* FLTL formulae *we use are given by the syntax:*

$$F, G, \ldots := c \mid F \wedge G \mid \neg F \mid \exists x F \mid \ominus F \mid \circ F \mid \Box F.$$

where c is a predicate symbol in $\mathcal{L}$.

The modalities $\ominus F, \circ F$ and $\Box F$ state, respectively, that F holds *previously, next* and *always*. We shall use $\forall x F$ for $\neg\exists x \neg F$, and $\Diamond F$ as an abbreviation of $\neg\Box\neg F$. Intuitively, $\Diamond F$ means that F eventually has to hold.

We say that F is a *state formula* if F does not have occurrences of temporal modalities.

2.4.1 Semantics of FLTL.

As done in Model Theory, the non-logical symbols of $\mathcal{L}$ (predicate, function and constant symbols) are given meaning in an underlying $\mathcal{L}$-structure, or $\mathcal{L}$-*model*, $\mathcal{M}(\mathcal{L}) = (\mathcal{I}, \mathcal{D})$. This means, they are interpreted via $\mathcal{I}$ as relations over a domain $\mathcal{D}$ of the corresponding arity.

States and Interpretations A *state* s is a mapping assigning to each variable x in $\mathcal{L}$ a value $s[x]$ in $\mathcal{D}$. This interpretation is extended to $\mathcal{L}$-expressions in the usual way, for example, $s[f(x)] = \mathcal{I}(f)(s[x])$. We write $s \models_{\mathcal{M}(\mathcal{L})} c$ if and only if c is true with respect to s in $\mathcal{M}(\mathcal{L})$.

The state s is said to be an x-variant of s' iff $s'[y] = s[y]$ for each $y \neq x$. This is, s and s' are the same except possibly for the value of the variable x.

We shall use $\sigma, \sigma', \ldots$ to range over infinite sequences of states. We say that σ is an x-variant of σ' iff for each $i \geq 0$, $\sigma(i)$ (the i-th state in σ) is an x-variant of $\sigma'(i)$.

Flexible and Rigid Variables. The set of variables is partitioned into *rigid* and *flexible*. For the rigid variables, each state σ must satisfy the *rigidity* condition: If x is rigid then for all s, s' in σ $s[x] = s'[x]$. If x is a flexible variable then different states in σ may assign different values to x.

Definition 2.4.2 (FLTL Semantics)**.** *We say that σ* satisfies *F in an $\mathcal{L}$-structure $\mathcal{M}(\mathcal{L})$, written $\sigma \models_{\mathcal{M}(\mathcal{L})} F$, if and only if $\langle \sigma, 0\rangle \models_{\mathcal{M}(\mathcal{L})} F$ where:*

$$\begin{array}{ll}
\langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} \texttt{true} & \\
\langle \sigma, i\rangle \not\models_{\mathcal{M}(\mathcal{L})} \texttt{false} & \\
\langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} c & \textit{iff } \sigma(i) \models_{\mathcal{M}(\mathcal{L})} c \\
\langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} \neg F & \textit{iff } \langle \sigma, i\rangle \not\models_{\mathcal{M}(\mathcal{L})} F \\
\langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} F \wedge G & \textit{iff } \langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} F \textit{ and } \langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} G \\
\langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} \ominus F & \textit{iff } i > 0 \textit{ and } \langle \sigma, i-1\rangle \models_{\mathcal{M}(\mathcal{L})} F \\
\langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} \circ F & \textit{iff } \langle \sigma, i+1\rangle \models_{\mathcal{M}(\mathcal{L})} F \\
\langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} \Box F & \textit{iff for all } j \geq i, \langle \sigma, j\rangle \models_{\mathcal{M}(\mathcal{L})} F \\
\langle \sigma, i\rangle \models_{\mathcal{M}(\mathcal{L})} \exists x F & \textit{iff for some } x\textit{-variant } \sigma' \textit{ of } \sigma, \langle \sigma', i\rangle \models_{\mathcal{M}(\mathcal{L})} F
\end{array}$$

We say that F *is* valid *in* $\mathcal{M}(\mathcal{L})$ *if and only if for all* σ, $\sigma \models_{\mathcal{M}(\mathcal{L})} F$. *$F$ is said to be* valid *if F is valid for every model* $\mathcal{M}(\mathcal{L})$.

Chapter 3

Syntax and Operational Semantics

This chapter introduces the syntax of the `utcc` calculus as well as its operational semantics. In the same lines of `tcc` [Saraswat 1994, Nielsen 2002a], the operational semantics of `utcc` is given by an internal and an observable transition relation. The first one describes the evolution of processes during a time unit. The second one describes how given an input from the environment, a process reacts outputting the final store obtained from a finite number of internal reductions. This defines the input-output behavior of a process. We shall also discuss some technical problems the abstraction construct of `utcc` poses in the operational semantics. Namely, we show that there exist processes that exhibit infinitely many internal reductions thus never producing an output. We shall deal with these termination problems in the next chapter where we present a symbolic semantics for `utcc`.

3.1 Constraint Systems

Concurrent Constraint programming (CCP) based calculi are parametric in a *constraint system* [Saraswat 1993] that specifies the basic constraints agents can tell or ask during execution. In this section we recall the definition of these systems.

A constraint represents a piece of (partial) information upon which processes may act. A constraint system then provides a signature from which constraints can be built. Furthermore, the constraint system provides an *entailment* relation ($\models$) specifying interdependencies between constraints. Intuitively, $c \models d$ means that the information d can be deduced from the information represented by c. For example, $x > 60 \models x > 42$.

Formally, we can set up the notion of constraint system by using First-Order Logic as in [Smolka 1994, Nielsen 2002a]. Let us suppose that Σ is a signature (i.e., a set of constant, function and predicate symbols) and that Δ is a consistent first-order theory over Σ (i.e., a set of sentences over Σ having at least one model). Constraints can be thought of as first-order formulae over Σ. Consequently, the entailment relation $\models_\Delta$ is defined as follows: $c \models_\Delta d$ if the implication $c \Rightarrow d$ is valid in Δ. This gives us a simple and general formalization of the notion of constraint system as a pair (Σ, Δ).

Definition 3.1.1 (Constraint System)**.** *A constraint system is as a pair* (Σ, Δ) *where* Σ *is a signature of constant, function and predicate symbols, and* Δ *is a first-order theory over* Σ *(i.e., a set of first-order sentences over* Σ *having at least one model).*

Given a constraint system (Σ, Δ), let $\mathcal{L}$ be its underlying first-order language with variables $x, y, \ldots$, and logic symbols $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall$, `true` and `false`. *Constraints*, denoted by $a, b, c, d, \ldots$ are first-order formulae over $\mathcal{L}$.

We say that c *entails* d in Δ, written $c \models_\Delta d$, iff $(c \Rightarrow d) \in \Delta$ (i.e., iff $c \Rightarrow d$ is true in all models of Δ). We shall omit "Δ" in $\models_\Delta$ when $\Delta = \emptyset$. Furthermore, we say that c is equivalent to d, written $c \equiv d$, iff $c \models_\Delta d$ and $d \models_\Delta c$.

Henceforth we shall use the following notation.

Notation 3.1.1 (Constraints and Equivalence)**.** *Henceforth,* $\mathcal{C}$ *denotes the set of constraints modulo* $\equiv$ *in the underlying constraint system. So, we write* $c = d$ *iff* c *and* d *are*

in the same ($\equiv$) class. Furthermore, whenever we write expressions such as $c = (x = y)$ we mean that c is (equivalent to) the constraint $x = y$.

Substitutions and Terms. Along this dissertation we shall use the following conventions for substitutions and terms.

Convention 3.1.1 (Terms and Substitutions)**.** *Let $\mathcal{T}$ be the set of terms induced by the signature Σ of the constraint system with typical elements $t, t', \ldots$ We use $\vec{t}$ for a sequence of terms $t_1, \ldots, t_n$ with length $|\vec{t}| = n$. If $|\vec{t}| = 0$ then $\vec{t}$, the empty sequence of terms, is written as ε. Given $\vec{t} = t_1.t_2....t_n$ and $\vec{t'} = t'_1.t'_2....t'_m$, we shall use $\vec{t};\vec{t'}$ to denote the sequence of terms $t_1.t_2....t_n.t'_1.t'_2....t'_m$.*

We shall use $\doteq$ to denote syntactic term equivalence (e.g. $x \doteq x$ and $x \neq y$). We write $\vec{x} \neq \vec{t}$ to denote $\bigvee_{1 \leq i \leq |\vec{x}|} x_i \neq t_i$. If $|\vec{x}| = 0$, $\vec{x} \neq \vec{t}$ is defined as `false`*.*

We use $c[\vec{t}/\vec{x}]$, where $|\vec{t}| = |\vec{x}|$ and x_i's are pairwise distinct, to denote c in which the free occurrences of x_i have been replaced with t_i. The substitution $[\vec{t}/\vec{x}]$ will be similarly applied to other syntactic entities.

We say that $\vec{t}$ is admissible for $\vec{x}$, notation $adm(\vec{x}, \vec{t})$, if $|\vec{x}| = |\vec{t}|$ and for all $i, j \in \{1, ..., |\vec{x}|\}$, $x_i \neq t_j$. If $|\vec{x}| = |\vec{t}| = 0$ then trivially $adm(\vec{x}, \vec{t})$. Similarly, we say that the substitution $[\vec{t}/\vec{x}]$ is admissible iff $adm(\vec{x}, \vec{t})$.

Basic Constraints and Processes. As traditionally done in CCP-based languages [Saraswat 1993, Smolka 1994, Fages 1998], processes will only be allowed to tell or ask basic constraints defined as follows.

Definition 3.1.2 (Basic Constraints)**.** *Let $p(\cdot)$ be a predicate symbol of arity $|\vec{t}|$. We say that c is a* basic constraint *iff c can be generated from the following syntax:*

$$c \;:=\; p(\vec{t}) \;\mid\; c \wedge c$$

3.2 Timed CCP (`tcc`)

In the CCP model, the information in the store evolves *monotonically*, i.e., once a constraint is added it cannot be removed. This condition has been relaxed by considering temporal extensions of CCP such as `tcc` [Saraswat 1994]. In `tcc`, time is conceptually divided into *time intervals* (or *time units*). In a particular time interval, a CCP process P gets an input c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines a residual process Q which is then executed in the next time interval. The resulting store d *is not automatically transferred* to the next time interval.

This view of reactive computation is particularly appropriate for programming reactive systems in the sense of Synchronous Languages [Berry 1992], i.e., systems that react continuously with the environment at a rate controlled by the environment.

Following the notation in [Nielsen 2002a], the syntax of `tcc` is as follows.

Definition 3.2.1 (Syntax of `tcc`)**.** *Processes $P, Q, \ldots$ in `tcc` are built from basic constraints in the underlying constraint system by the following syntax:*

$$\begin{aligned} P, Q := \;& \mathbf{skip} \mid \mathbf{tell}(c) \mid \mathbf{when}\ c\ \mathbf{do}\ P \mid P \parallel Q \mid \\ & (\mathbf{local}\,\vec{x}; c)\,P \mid \mathbf{next}\,P \mid \mathbf{unless}\ c\ \mathbf{next}\,P \mid !\,P \end{aligned}$$

with the variables in $\vec{x}$ being pairwise distinct.

The process **skip** does nothing thus representing *inaction*. The process **tell**(c) adds c to the store in the current time interval, thus making it available to the other processes. The process **when** c **do** P *asks* if c can be deduced from the store. If so, it behaves as P. In other case, it remains blocked until the store contains at least as much information as c. The process $P \parallel Q$ denotes P and Q running concurrently during the current time interval possibly "communicating" via the common store. Given a finite set of indexes $I = \{i_1, i_2, ..., i_n\}$, we shall use $\prod_{i \in I} P_i$ to denote the parallel composition $P_{i_1} \parallel P_{i_2} \parallel ... \parallel P_{i_n}$.

Hiding on a set of variables $\vec{x}$ is enforced by the process $(\mathbf{local}\, \vec{x}; c)\, P$. It behaves like P, except that all the information on the variables $\vec{x}$ produced by P can only be seen by P and the information on the global variables in $\vec{x}$ produced by other processes cannot be seen by P. The local information on $\vec{x}$ produced by P corresponds to the constraint c representing a *local store*. When $c = \mathtt{true}$, we shall simply write $(\mathbf{local}\, \vec{x})\, P$ instead of $(\mathbf{local}\, \vec{x}; \mathtt{true})\, P$.

The process $(\mathbf{local}\, \vec{x}; c)\, P$ *binds* the variables $\vec{x}$ in P. We use $bv(P)$ and $fv(P)$, to denote respectively the set of *bound variables* and *free variables* in P.

Timed Constructs. The *unit-delay* $\mathbf{next}\, P$ executes P in the next time interval. The *negative ask* **unless** c $\mathbf{next}\, P$ is also a unit-delay but P is executed in the next time unit if and only if c *is not* entailed by the final store at the current time interval. This can be viewed as a (weak) time-out: It waits one time unit for a piece of information c to be present and if it is not, it triggers activity in the next time interval. The process P must be guarded by a next process to avoid paradoxes such as a program that requires a constraint to be present at an instant only if it is not present at that instant (see [Saraswat 1994]).

Finally, the *replication* $!P$ means $P \parallel \mathbf{next}\, P \parallel \mathbf{next}^2 P$..., i.e. unboundedly many copies of P but one at a time.

Remark 3.2.1. *Notice that in general* $Q =$ **unless** c $\mathbf{next}\, P$ *does not behave the same as* $Q' =$ **when** $\neg c$ **do next** P. *This can be explained from the fact that* $d \not\models_\Delta c$ *does not imply* $d \models_\Delta \neg c$. *Let for example* Δ *be the axioms of Peano arithmetic and assume* $d =$ "$x > 0$" *and* $c =$ "$x = 42$". *We have both,* $x > 0 \not\models_\Delta x \neq 42$ *and* $x > 0 \not\models_\Delta x = 42$. *Then, the process* P *is executed in* Q *but it is precluded from execution in* Q'.

3.3 Abstractions and Universal Timed CCP

In [Saraswat 1994, Valencia 2005], `tcc` processes were shown to be finite-state. This suggests they cannot be used to describe infinite-state behaviors like those arising from mobile systems such as Security Protocols which is one of the application domains in this dissertation. Here mobility is understood in the sense of the π-calculus [Milner 1992b, Sangiorgi 2001], i.e., communication of (private) variables or names.

Let us take for example a predicate (constraint) of the form $\mathtt{out}(\cdot)$ and let $P =$ **when** $\mathtt{out}(x)$ **do** R. We notice that under input $\mathtt{out}(42)$, P does not execute R since $\mathtt{out}(42)$ does not entail $\mathtt{out}(x)$ (i.e. $\mathtt{out}(42) \not\models \mathtt{out}(x)$). The issue here is that x is a free-variable and hence does not act as a formal parameter (or place holder) for every term t such that $\mathtt{out}(t)$ is entailed by the store.

To model mobile behavior, `utcc` replaces the `tcc` ask operation **when** c **do** P with a more general parametric ask construction, namely $(\mathbf{abs}\, \vec{x}; c)\, P$. This process can be viewed as a λ-*abstraction* of the process P on the variables $\vec{x}$ under the constraint (or with the *guard*) c. Intuitively, $Q = (\mathbf{abs}\, \vec{x}; c)\, P$ performs $P[\vec{t}/\vec{x}]$ (i.e. P with the free occurrences of $\vec{x}$ replaced with $\vec{t}$) in the current time interval for *all the terms* $\vec{t}$ s.t $c[\vec{t}/\vec{x}]$ is entailed by the current store. For example, $P = (\mathbf{abs}\, x; \mathtt{out}(x))\, R$ under input $\mathtt{out}(42)$ executes $R[42/x]$.

This way, from a programming language perspective, while we can see the variables $\vec{x}$ in $(\mathbf{local}\,\vec{x}; c)\,P$ as the local variables of P, we can see $\vec{x}$ in $(\mathbf{abs}\ \vec{x}; c)\,P$ as the formal parameters of the process P.

Definition 3.3.1 (`utcc` Processes)**.** *The* `utcc` *processes result from replacing* **when** c **do** P *by* $(\mathbf{abs}\ \vec{x}; c)\,P$ *in the syntax of Definition 3.2.1. The variables in* $\vec{x}$ *are assumed to be pairwise distinct.*

The process $Q = (\mathbf{abs}\ \vec{x}; c)\,P$ binds the variables $\vec{x}$ in P and c. The sets of free and bound variables, $fv(\cdot)$ and $bv(\cdot)$ respectively, are extended accordingly. Furthermore Q evolves into **skip** at the end of the time unit, i.e., abstractions are not persistent when passing from one time unit to the next one.

Notation 3.3.1. *Recall that* ε *denotes the empty vector of variables. We shall write* **when** c **do** P *instead of the the empty abstraction* $(\mathbf{abs}\ \varepsilon; c)\,P$.

3.3.1 Recursion in `utcc`

In some of the applications of this dissertation we shall use parametric process definitions of the form

$$p(\vec{y}) \stackrel{\mathtt{def}}{=} P$$

where $\vec{y}$ is a set of pairwise distinct variables, p is the recursive definition name and P can recursively call $p(\cdot)$.

As in the π-calculus [Milner 1992c], we do not want, when unfolding recursive calls of p, the free variables of P to get captured in the lexical-scope of a bound-variable in P. In other words, we want static scoping rather than dynamic scoping. Then, following [Nielsen 2002a, Milner 1992c] we assume that $fv(P) \subseteq \vec{y}$.

Intuitively, a call of a recursive definition of the form $p(\vec{t})$ must execute the process P substituting $\vec{y}$ (the formal parameters) in P by $\vec{t}$ (the actual parameters). This clearly resembles the idea of an abstraction in `utcc`. The following encoding of recursion reflects this intuition.

Definition 3.3.2 (Recursive Definitions in `utcc`)**.** *Assume a recursive definition of the form* $p(\vec{y}) \stackrel{\mathtt{def}}{=} P$. *We add to the constraint system under consideration an uninterpreted predicate* $call_p(\cdot)$ *of arity* $|\vec{y}|$. *The process definition and calls can be encoded as follows:*

- *For the process definitions:* $\ulcorner p(\vec{y}) \stackrel{\mathtt{def}}{=} P \urcorner =! (\mathbf{abs}\ \vec{y}; call_p(\vec{y}))\,\widehat{P}$
  where $\widehat{P}$ *is the process obtained by replacing in* P *any call* $p(\vec{x})$ *by* $\mathbf{tell}(call_p(\vec{x}))$.
- *Analogously, the call* $p(\vec{x})$ *in all its other occurrences is replaced by* $\mathbf{tell}(call_p(\vec{x}))$.

From now on, we shall freely use parametric recursive definitions taking into account that they can be straightforwardly encoded as abstractions.

Remark 3.3.1. *Assume a recursive definition of the form* $p(x) \stackrel{\mathtt{def}}{=} P$. *In a programming language with recursion, if several calls of* $p(x)$ *are executed using the same actual parameter* t, *each call will spawn the execution of* $P[t/x]$. *Note that this is not the case in the encoding above. The issue here is that the abstraction modeling the recursive definition reduces to* $(\mathbf{abs}\ x; call_p(x) \wedge x \neq t)\,P$. *Therefore, a second call of* $p(t)$ *(adding* $call_p(t)$ *to the current store) does not spawn* $P[t/x]$ *again. This can be also explained from the fact that adding twice the constraint* $call_p(t)$ *to the store has the same effect that adding it only once.*

From this, notice also that a recursive definition of the form $p(x) \stackrel{\mathtt{def}}{=} p(x)$ *does not cause divergent computations in the encoding above.*

3.4 Structural Operational Semantics

The structural operational semantics (SOS) [Plotkin 1981] of `utcc` considers *transitions* between process-store *configurations* $\langle P, c\rangle$ with stores represented as constraints and processes quotiented by the structural congruence $\equiv$ in Definition 3.4.1. We shall use $\gamma, \gamma', \ldots$ to range over configurations.

Definition 3.4.1 (Structural Congruence)**.** *Let $\equiv$ be the smallest congruence satisfying:*

1. $P \equiv Q$ *if they differ only by a renaming of bound variables (alpha-conversion).*
2. $P \parallel \mathbf{skip} \equiv P$
3. $P \parallel Q \equiv Q \parallel P$
4. $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$
5. $P \parallel (\mathbf{local}\,\vec{x}; c)\,Q \equiv (\mathbf{local}\,\vec{x}; c)\,(P \parallel Q)$ *if* $\vec{x} \notin fv(P)$ *(Scope Extrusion)*
6. $(\mathbf{local}\,\vec{x}; c)\,(\mathbf{local}\,\vec{y}; d)\,P \equiv (\mathbf{local}\,\vec{x}; \vec{y}\,; c \wedge d)\,P \quad$ *if* $\vec{x} \cap \vec{y} = \emptyset$ *and* $\vec{y} \notin fv(c)$.

We extend $\equiv$ by decreeing that $\langle P, c\rangle \equiv \langle Q, d\rangle$ iff $P \equiv Q$ and $c \equiv d$.

Notice that we have used the same symbol for logical equivalence (Section 3.1) and for structural congruence of processes. The meaning of $\equiv$ shall be then understood according to its operands.

Internal and Observable Transitions. The SOS transitions are given by the relations $\longrightarrow$ and $\Longrightarrow$ in Table 3.1. The *internal* transition $\langle P, d\rangle \longrightarrow \langle P', d'\rangle$ should be read as "P with store d reduces, in one internal step, to P' with store d' ". The *observable transition* $P \xrightarrow{(c,d)} R$ should be read as "P on input c, reduces in one *time unit* to R and outputs d". The observable transitions are obtained from finite sequences of internal transitions.

Let us describe the internal reduction rules in Table 3.1.

- The rule $\mathrm{R}_{\mathrm{TELL}}$ says that the process $\mathbf{tell}(c)$ adds c to the current store d, via conjunction, and evolves into **skip**.
- The rule $\mathrm{R}_{\mathrm{PAR}}$ is the standard interleaving rule for parallel composition: If P may evolve into P', this reduction also takes place when running in parallel with other process Q.
- Let $Q = (\mathbf{local}\,\vec{x}; c)\,P$ in Rule $\mathrm{R}_{\mathrm{LOC}}$. The global store is d and the local store is c. We distinguish between the *external* (corresponding to Q) and the *internal* point of view (corresponding to P). From the internal point of view, the information about $\vec{x}$, possibly appearing in the "global" store d, cannot be observed. Thus, before reducing P we first hide the information about $\vec{x}$ that Q may have in d by existentially quantifying $\vec{x}$ in d. Similarly, from the external point of view, the observable information about $\vec{x}$ that the reduction of internal agent P may produce (i.e., c') cannot be observed. Thus we hide it by existentially quantifying $\vec{x}$ in c' before adding it to the global store. Additionally, we make c' the new private store of the evolution of the internal process.
- Since the process $P = \mathbf{unless}\ c\ \mathbf{next}\,Q$ executes Q in the next time unit only if the final store at the current time unit does not entail c, in the rule $\mathrm{R}_{\mathrm{UNL}}$ P evolves into **skip** if the current store d entails c.

$$\mathrm{R_{TELL}}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \wedge c\rangle}$$

$$\mathrm{R_{PAR}}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle}$$

$$\mathrm{R_{LOC}}\ \frac{\langle P, c \wedge (\exists \vec{x} d)\rangle \longrightarrow \langle P', c' \wedge (\exists \vec{x} d)\rangle}{\langle (\mathbf{local}\, \vec{x}; c)\, P, d\rangle \longrightarrow \langle (\mathbf{local}\, \vec{x}; c')\, P', d \wedge \exists \vec{x} c'\rangle}$$

$$\mathrm{R_{UNL}}\ \frac{d \models_\Delta c}{\langle \mathbf{unless}\ c\ \mathbf{next}\ P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle}$$

$$\mathrm{R_{REP}}\ \frac{}{\langle !\, P, d\rangle \longrightarrow \langle P \parallel \mathbf{next}\, !\, P, d\rangle}$$

$$\mathrm{R_{ABS}}\ \frac{d \models_\Delta c[\vec{t}/\vec{x}] \qquad [\vec{t}/\vec{x}] \text{ is admissible.}}{\langle (\mathbf{abs}\ \vec{x}; c)\, P, d\rangle \longrightarrow \langle P[\vec{t}/\vec{x}] \parallel (\mathbf{abs}\ \vec{x}; c \wedge \vec{x} \neq \vec{t}\,)\, P, d\rangle}$$

$$\mathrm{R_{STR}}\ \frac{\gamma_1 \longrightarrow \gamma_2}{\gamma_1' \longrightarrow \gamma_2'} \text{ if } \gamma_1 \equiv \gamma_1' \text{ and } \gamma_2 \equiv \gamma_2'$$

$$\mathrm{R_{OBS}}\ \frac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow}{P \xRightarrow{(c,d)} F(Q)}$$

Table 3.1: Internal and observable reductions. $\equiv$ and F are given in Definitions 3.4.1 and 3.4.2 respectively. $\neq$ and admissibility of $[\vec{t}/\vec{x}]$ are defined in Convention 3.1.1.

- Rule $\mathrm{R_{REP}}$ dictates that the process $!\,P$ produces a copy of P at the current time unit, and then persists in the next time unit.
- Rule $\mathrm{R_{ABS}}$ describes the behavior of $P = (\mathbf{abs}\ \vec{x}; c)\, Q$. Recall that $adm(\vec{x}, \vec{t})$ means that none of the variables in $\vec{x}$ is syntactically equal to an element in $\vec{t}$ and then, the substitution $[\vec{t}/\vec{x}]$ is admissible. If the current store entails $c[\vec{t}/\vec{x}]$, then the process $P[\vec{t}/\vec{x}]$ is executed. Additionally, the abstraction persists in the current time interval to allow other potential replacements of $\vec{x}$ in P. Notice that c is augmented with $\vec{x} \neq \vec{t}$ to avoid executing $P[\vec{t}/\vec{x}]$ again.
- Rule $\mathrm{R_{STR}}$ says that structurally congruent configurations have the same reductions.

Observable Transition and Future Function. The seemingly missing rules for the processes **next** P and **unless** c **next** P (when c cannot be entailed from the current store) are given by the rule $\mathrm{R_{OBS}}$. This rule says that an observable transition from P labeled with (c, d) is obtained from a terminating sequence of internal transitions from $\langle P, c\rangle$ to $\langle Q, d\rangle$. The process R to be executed in the next time interval is equivalent to $F(Q)$ (the "future" of Q). The process $F(Q)$ is obtained by removing from Q abstractions and any local information that has been stored in Q, and by "unfolding" the sub-terms within **next**

and **unless** expressions. More precisely:

Definition 3.4.2. *Let F be a partial function defined as:*

$$F(P) = \begin{cases} \mathbf{skip} & \text{if } P = \mathbf{skip} \\ \mathbf{skip} & \text{if } P = (\mathbf{abs}\ \vec{x}; c)\, Q \\ F(P_1) \parallel F(P_2) & \text{if } P = P_1 \parallel P_2 \\ (\mathbf{local}\, \vec{x})\, F(Q) & \text{if } P = (\mathbf{local}\, \vec{x}; c)\, Q \\ Q & \text{if } P = \mathbf{next}\, Q \\ Q & \text{if } P = \mathbf{unless}\ c\ \mathbf{next}\, Q \end{cases}$$

Remark 3.4.1. *Notice that F can be defined as a partial function since whenever we need to apply F to a P, the processes of the form* $\mathbf{tell}(c)$ *and* $!\,Q$ *must be occur within a* **next** *or* **unless** *expression.*

For the sake of presentation, in the sequel we assume the following convention.

Convention 3.4.1 (Local Information)**.** *Given an observable transition $P \xrightarrow{(c,c')} Q$, we assume that for all process of the form* $(\mathbf{local}\, \vec{x}; c)\, R$ *in P, $c =$* `true`*. Then, we simply write* $(\mathbf{local}\, \vec{x})\, R$*. Note that this is not a loss of generality since for any c, the process* $(\mathbf{local}\, \vec{x}; c)\, R$ *can be written as* $(\mathbf{local}\, \vec{x})\, (R \parallel \mathbf{tell}(c))$*.*

To conclude this section, we define the size of a process. We shall use this measure in some of the proofs in this dissertation.

Definition 3.4.3 (Size of a process)**.** *Given a* `utcc` *process P, we define the size of P as*

$$M(P) = \begin{cases} 0 & \text{if } P = \mathbf{skip} \\ 1 & \text{if } P = \mathbf{tell}(c) \\ 1 + M(P') & \text{if } P = (\mathbf{abs}\ \vec{x}; c)\, P' \\ M(Q) + M(R) & \text{if } P = Q \parallel R \\ 1 + M(P') & \text{if } P = (\mathbf{local}\, \vec{x}; c)\, P' \\ 1 + M(P') & \text{if } P = \mathbf{next}\, P' \\ 1 + M(P') & \text{if } P = \mathbf{unless}\ c\ \mathbf{next}\, P' \\ 1 + M(P') & \text{if } P =\, !\, P' \end{cases}$$

3.5 Properties of the Internal Transitions

In this section we study some simple but fundamental properties of the internal reduction relation that we shall use in the forthcoming results.

The first property states that the store can only be augmented.

Lemma 3.5.1 (Internal Extensiveness)**.** *If $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$ then $d \models_\Delta c$.*

Proof. The proof proceeds by a simple induction on the inference of $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$. □

Augmenting the store may increase the potentiality of internal reductions, that is, the number of possible internal transitions. The following lemma states that any configuration $\langle Q, e\rangle$ obtained from $\langle P, d\rangle$ can also be obtained from $\langle P, c\rangle$, where c entails d and c is weaker than e.

Lemma 3.5.2 (Internal Potentiality)**.** *If $e \models_\Delta c \models_\Delta d$ and $\langle P, d\rangle \longrightarrow \langle Q, e\rangle$ then $\langle P, c\rangle \longrightarrow \langle Q, e\rangle$*

Proof. Assume that $e \models_\Delta c \models_\Delta d$. We proceed by induction on the inference of $\langle P, d\rangle \longrightarrow \langle Q, e\rangle$. We consider only the case for the rule $\mathrm{R}_{\mathrm{ABS}}$. The other cases are easy. If the rule $\mathrm{R}_{\mathrm{ABS}}$ was used in the derivation $\langle P, d\rangle \longrightarrow \langle Q, e\rangle$, it must be the case that $P \equiv (\mathbf{abs}\ \vec{x}; c')\ P'$ and there exists a term $\vec{t}$ such that $d \models_\Delta c'[\vec{t}/\vec{x}]$. Hence, if $c \models_\Delta d$ then $c \models_\Delta c'[\vec{t}/\vec{x}]$ and we conclude $\langle P, c\rangle \longrightarrow \langle Q, e\rangle$. □

Finally we state a lemma which resembles a fixed point property.

Lemma 3.5.3 (Internal Restartability)**.** *Whenever* $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$, $\langle P, d\rangle \longrightarrow \langle Q, d\rangle$.

Proof. Assume that $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$. Then from Lemma 3.5.1 $d \models_\Delta c$. The result follows from Lemma 3.5.2. □

3.6 Mobility in utcc

Mobility in utcc is obtained from the interplay between the abstractions and the local operators. Recall that here mobility is understood in the sense of the π-calculus [Milner 1992b, Sangiorgi 2001], i.e, communication of (private) variables or names. Let us illustrate this by modeling in utcc the example presented in Section 2.2.3.

Example 3.6.1 (Scope Extrusion)**.** *Let* Σ *be a signature with the unary predicates* $\mathtt{out}_1, \mathtt{out}_2, \ldots$ *and a constant* 0*. Let* $\Delta = \emptyset$ *and* P, Q *be processes defined as follows*

$$\begin{array}{lcl} P & = & (\mathbf{abs}\ y; \mathtt{out}_1(y))\, \mathbf{tell}(\mathtt{out}_2(y)) \\ Q & = & (\mathbf{local}\ z)\, (\mathbf{tell}(\mathtt{out}_1(z)) \parallel \mathbf{when}\ \mathtt{out}_2(z)\ \mathbf{do}\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}_2(0))) \end{array}$$

Intuitively, if a link y *is sent on channel* $\mathtt{out}_1$*,* P *forwards it on* $\mathtt{out}_2$*. Now,* Q *sends its private link* z *on* $\mathtt{out}_1$ *and if it gets it back on* $\mathtt{out}_2$ *it outputs* 0 *on* $\mathtt{out}_2$*.*

Let $\gamma = \langle P \parallel Q, \mathtt{true}\rangle$*. Using the rules in the Table 3.1 we can verify that* γ *evolves into a configuration including the process* $\mathbf{next}\, \mathbf{tell}(\mathtt{out}_2(0))$*:*

$$\begin{array}{lll} \gamma & \longrightarrow^* & \langle (\mathbf{local}\ z)\, (\mathbf{tell}(\mathtt{out}_1(z)) \parallel \mathbf{when}\ \mathtt{out}_2(z)\ \mathbf{do}\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}_2(0)) \parallel P) \\ & & \quad , \mathtt{true}\rangle \text{ — by structural congruence (scope extrusion) and rule } \mathrm{R}_{\mathrm{STR}} \\ & \longrightarrow^* & \langle (\mathbf{local}\ z; \mathtt{out}_1(z))\, (\mathbf{skip} \parallel \mathbf{when}\ \mathtt{out}_2(z)\ \mathbf{do}\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}_2(0)) \parallel P) \\ & & \quad , \exists_z(\mathtt{out}_1(z))\rangle \\ & \longrightarrow^* & \langle (\mathbf{local}\ z; \mathtt{out}_1(z))\, (\mathbf{when}\ \mathtt{out}_2(z)\ \mathbf{do}\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}_2(0)) \parallel P' \\ & & \qquad \parallel \mathbf{tell}(\mathtt{out}_2(z))) \ , \ \exists_z(\mathtt{out}_1(z))\rangle \text{ — by Rule } \mathrm{R}_{\mathrm{ABS}} \\ & \longrightarrow^* & \langle (\mathbf{local}\ z; \mathtt{out}_1(z) \wedge \mathtt{out}_2(z))\, (\mathbf{next}\, \mathbf{tell}(\mathtt{out}_2(0)) \parallel P') \\ & & \quad , \exists_z(\mathtt{out}_1(z) \wedge \mathtt{out}_2(z))\rangle \not\longrightarrow \end{array}$$

where $P' = (\mathbf{abs}\ y; \mathtt{out}_1(y) \wedge y \neq z)\, \mathbf{tell}(\mathtt{out}_2(y))$*. Let* $d = \exists_z(\mathtt{out}_1(z) \wedge \mathtt{out}_2(z))$*. We then conclude* $P \parallel Q \xrightarrow{(\mathtt{true}, d)} \mathbf{tell}(\mathtt{out}_2(0))$ *as expected.* □

Observation 3.6.1 (Scope Extrusion)**.** *Notice that in the derivation above, the Equation (5) in the structural congruence (Definition 3.4.1) is used in the first step to extrude the scope of the local variable* z *defined by the process* Q*. This way, the abstraction in* P *is able to substitute* z *for* y *under the same local environment.*

The reader may have noticed that in the previous example the number of communication channels is determined by the number of predicates of the form $\mathtt{out}_i$ in the underlying constraint system. Furthermore, they can only be seen as public channels since any process can send a datum on them. The next example uses binary predicates to provide for local channels as in the π-calculus.

Example 3.6.2 (Private Channels)**.** *Let Σ be a signature with a binary predicate* out *and the constant symbols $a, 0$. Let Δ be as in the Example 3.6.1 and P and Q be defined as:*

$$\begin{array}{lcl} P & = & (\mathbf{local}\, z)\,(\mathbf{tell}(\mathtt{out}(a,z)) \parallel (\mathbf{abs}\ y; \mathtt{out}(z,y))\,\mathbf{tell}(\mathtt{out}(y,0))) \\ Q & = & (\mathbf{abs}\ x; \mathtt{out}(a,x))\,(\mathbf{local}\, z')\,(\mathbf{tell}(\mathtt{out}(x,z')) \parallel \mathbf{when}\ \mathtt{out}(z',0)\ \mathbf{do}\ R) \end{array}$$

Here the constant a can be seen as the name of a public channel that P and Q share to communicate each other their local names (or private channels) z and z' respectively. Once Q receives the local name z from P, it sends z' on channel z. Then, P receives the channel z' from Q and outputs on it the constant 0 triggering the execution of R in Q. To see this, let us show the internal reductions derived from the configuration $\gamma = \langle P \parallel Q, \mathtt{true}\rangle$:

$$\begin{array}{lcl} \gamma & \longrightarrow^* & \langle (\mathbf{local}\, z)\,(\mathbf{tell}(\mathtt{out}(a,z)) \parallel (\mathbf{abs}\ y; \mathtt{out}(z,y))\,(\mathbf{tell}(\mathtt{out}(y,0))) \parallel Q) \\ & & \quad , \ \mathtt{true}\rangle \\ & \longrightarrow^* & \langle (\mathbf{local}\, z; \mathtt{out}(a,z))\,((\mathbf{abs}\ y; \mathtt{out}(z,y))\,(\mathbf{tell}(\mathtt{out}(y,0))) \parallel Q) \\ & & \quad , \ \exists_z(\mathtt{out}(a,z))\rangle \\ & \longrightarrow^* & \langle (\mathbf{local}\, z, z'; \mathtt{out}(a,z))\,((\mathbf{abs}\ y; \mathtt{out}(z,y))\,(\mathbf{tell}(\mathtt{out}(y,0))) \\ & & \qquad\qquad \parallel Q' \parallel \mathbf{tell}(\mathtt{out}(z,z')) \\ & & \qquad\qquad \parallel \mathbf{when}\ \mathtt{out}(z',0)\ \mathbf{do}\ R) \\ & & \quad , \ \exists_z(\mathtt{out}(a,z))\rangle \\ & \longrightarrow^* & \langle (\mathbf{local}\, z, z'; \mathtt{out}(a,z) \wedge \mathtt{out}(z,z'))\,(P' \parallel \mathbf{tell}(\mathtt{out}(z',0)) \parallel Q' \parallel \\ & & \qquad\qquad \mathbf{when}\ \mathtt{out}(z',0)\ \mathbf{do}\ R) \\ & & \quad , \ \exists_{z,z'}(\mathtt{out}(a,z) \wedge \mathtt{out}(x,z'))\rangle \\ & \longrightarrow^* & \langle (\mathbf{local}\, z, z'; \mathtt{out}(a,z) \wedge \mathtt{out}(z,z') \wedge \mathtt{out}(z',0))\,(P' \parallel Q' \parallel R)\ ,\ d\rangle \not\longrightarrow \end{array}$$

where

$$\begin{array}{lcl} Q' & = & (\mathbf{abs}\ x; \mathtt{out}(a,x) \wedge x \neq z)\,(\mathbf{local}\, z')\,(\mathbf{tell}(\mathtt{out}(x,z')) \parallel \mathbf{when}\ \mathtt{out}(z',0)\ \mathbf{do}\ R) \\ P' & = & (\mathbf{abs}\ y; \mathtt{out}(z,y) \wedge y \neq z')\,\mathbf{tell}(\mathtt{out}(y,0)) \\ d & = & \exists_{z,z'}(\mathtt{out}(a,z) \wedge \mathtt{out}(x,z') \wedge \mathtt{out}(z',0)) \end{array}$$

As expected, the processes P and Q running in parallel reduce to a process where R is executed. □

In Chapter 8, for the applications to security, we shall use the communication pattern in Example 3.6.1. This communication pattern is akin to the version of the spi-calculus [Abadi 1997] in [Fiore 2001] where only a global (public) channel is considered. The intuition is that all the messages are sent through an untrusted network under the control of the spy. The advantage is that we do not require binary predicates as in Example 3.6.2. Furthermore, as was pointed out in [Hildebrandt 2009], the use of binary predicates to model communication channels allows agents to guess channel names by universal quantification (see related work in Section 3.9).

Unary predicates as communication channels are also used in this dissertation in Chapter 6 to encode Minsky machines into utcc. In this case, having a simple constraint system with only unary predicates is central to apply this encoding to prove the undecidability of the monadic fragment of first-order linear-time temporal logic (see Chapter 6).

3.7 Input-Output Behavior

In this section we define the notion of observable behavior in utcc. Furthermore, we show that utcc processes are deterministic, i.e., the outputs of a process are the same up to logical equivalence.

Reactive Observations. Consider the following sequence of observable transitions

$$P = P_1 \xrightarrow{(c_1,c'_1)} P_2 \xrightarrow{(c_2,c'_2)} P_3 \xrightarrow{(c_3,c'_3)} \dots$$

This sequence can be seen as the interaction of the system P with the environment. At the time unit i, the environment provides as input the constraint c_i and P responds with the final store c'_i. Let $\alpha = c_1.c_2.c_3...$ and $\alpha' = c'_1.c'_2.c'_3...$ be sequences of constraints. As observers, we can see that on input α, the process P responds with α'. We then regard (α, α') as a *reactive observation of* P. We shall call the set of reactive observations of P the *input-output behavior* of P.

When the sequence of inputs α is the sequence `true.true.true`..., we say that P outputs α' without the influence of any environment. Then, we say that α' is the *default output* of P.

Before stating formally the above notions of behavior, we require the following notation on sequences of constraints.

Notation 3.7.1. *We shall denote with $\mathcal{C}^\omega$ the set of infinite sequences of constraints with typical elements $\alpha, \alpha', \beta, \beta', \dots$. Given $c \in \mathcal{C}$, c^ω represents the sequence of constraints $c.c.c.\dots$. The i-th element in α is denoted by $\alpha(i)$. We shall write $\alpha \geq \alpha'$ whenever $\alpha(i) \models_\Delta \alpha'(i)$ for $i > 0$.*

Definition 3.7.1 (Input-Output Relation and Equivalences)**.** *Let P be a `utcc` process. Given $\alpha = c_1.c_2...$ and $\alpha' = c'_1.c'_2...$, we write $P \xrightarrow{(\alpha,\alpha')}$ whenever*

$$P = P_1 \xrightarrow{(c_1,c'_1)} P_2 \xrightarrow{(c_2,c'_2)} P_3 \xrightarrow{(c_3,c'_3)} \dots$$

The set $io(P) = \{(\alpha, \alpha') \mid P \xrightarrow{(\alpha,\alpha')}\}$ denotes the input-output *behavior of P. If $io(P) = io(Q)$ we say that P and Q are input-output equivalent and we write $P \sim^{io} Q$.*

We say that α' is the default output of P, denoted by $o(P)$, if $(\texttt{true}^\omega, \alpha') \in io(P)$. This means, P outputs α' without the influence of any (external) environment. Furthermore, if there exists $i > 0$ s.t. $\alpha'(i) \models_\Delta c$, we say that P eventually outputs c and we write $P \Downarrow^c$. Finally, if $o(P) = o(Q)$ we say that P and Q are output equivalent and we write $P \sim^o Q$.

Based on the rules of internal and observable transitions, we can make the following observation over the elements of the input-output relation.

Observation 3.7.1. *Let P be a process and α, α' be sequences of constraints such that $(\alpha, \alpha') \in io(P)$. Similar to `tcc` [Saraswat 1994], computations in `utcc` during a time unit progress via the monotonic accumulation of constraints (see Lemma 3.5.1). Then, for all $i > 0$, $\alpha'(i) \models_\Delta \alpha(i)$. Recall also that the final store at the end of the time unit is not automatically transferred to the next one. Therefore, it may be the case that $\alpha'(i) \not\models_\Delta \alpha'(i-1)$. Finally, constraints in α are provided by the environment as input to the system and then, they are not supposed to be related to each other.*

Determinism. Now we prove that `utcc` processes are *deterministic*, i.e., the outputs of a process are equivalent regardless the execution order of the parallel components.

We first need to state an important property of internal transitions, namely that of *confluence*.

Lemma 3.7.1 (Confluence)**.** *Suppose that $\gamma_0 \longrightarrow \gamma_1$, $\gamma_0 \longrightarrow \gamma_2$ and $\gamma_1 \not\equiv \gamma_2$. Then, there exists γ_3 such that $\gamma_1 \longrightarrow \gamma_3$ and $\gamma_2 \longrightarrow \gamma_3$.*

Proof. Given a configuration $\gamma = \langle P, c\rangle$ we define the size of γ as the size of P, $M(P)$ (see Definition 3.4.3). Suppose that $\gamma_0 \equiv \langle P, c_0\rangle \longrightarrow \gamma_1$, $\gamma_0 \longrightarrow \gamma_2$ and $\gamma_1 \not\equiv \gamma_2$. The proof proceeds by induction on the size of γ_0. From the assumption $\gamma_1 \not\equiv \gamma_2$, it must be the case that P is not a process of the form $\mathbf{tell}(c)$, $!\,P'$ or $\mathbf{unless}\ c\ \mathbf{next}\ P'$ since from those processes there is a unique possible transition modulo structural congruence.

For the case $P = Q \parallel R$, we have to consider three cases. Assume that $\gamma_1 \equiv \langle Q' \parallel R, c_1\rangle$ and $\gamma_2 \equiv \langle Q'' \parallel R, c_2\rangle$. We know by induction that if $\gamma_0' \equiv \langle Q, c_0\rangle \longrightarrow \gamma_1' \equiv \langle Q', c_1\rangle$ and $\gamma_0' \longrightarrow \gamma_2' \equiv \langle Q'', c_2\rangle$ then there exists $\gamma_3' \equiv \langle Q''', c_3\rangle$ such that $\gamma_1' \longrightarrow \gamma_3'$ and $\gamma_2' \longrightarrow \gamma_3'$. We conclude by noticing that $\gamma_1 \longrightarrow \gamma_3 \equiv \langle Q''' \parallel R, c_3\rangle$ and $\gamma_2 \longrightarrow \gamma_3$ by rule $\mathrm{R}_{\mathrm{PAR}}$. The case when R has two possible transitions is similar to the previous one. Now assume that $\gamma_1 \equiv \langle Q' \parallel R, c_0 \wedge c_1\rangle$ and $\gamma_2 \equiv \langle Q \parallel R', c_0 \wedge c_2\rangle$. Then, by Lemma 3.5.1 we have $\gamma_3 \equiv \langle Q' \parallel R', c_0 \wedge c_1 \wedge c_2\rangle$.

Finally, let $\gamma_0 \equiv \langle P, c_0\rangle$ with $P = (\mathbf{abs}\ \vec{x}; c)\, Q$. One can verify that $\gamma_1 \equiv \langle P_1, c_0\rangle$ where P_1 takes the form $(\mathbf{abs}\ \vec{x}; c \wedge \vec{x} \neq \vec{t_1})\, Q \parallel Q[\vec{t_1}/\vec{x}]$ and $\gamma_2 \equiv \langle P_2, c_0\rangle$ where P_2 takes the form $(\mathbf{abs}\ \vec{x}; c \wedge \vec{x} \neq \vec{t_2})\, Q \parallel Q[\vec{t_2}/\vec{x}]$ for some terms $\vec{t_1}$ and $\vec{t_2}$. From the assumption $\gamma_1 \not\equiv \gamma_2$, it must be the case that $\vec{t_1} \neq \vec{t_2}$. Let $\gamma_3 \equiv \langle P_3, c_0\rangle$ where $P_3 = (\mathbf{abs}\ \vec{x}; c \wedge \vec{x} \neq \vec{t_1} \wedge \vec{x} \neq \vec{t_2})\, Q \parallel Q[\vec{t_1}/\vec{x}] \parallel Q[\vec{t_2}/\vec{x}]$. Clearly $\gamma_1 \longrightarrow \gamma_3$ and $\gamma_2 \longrightarrow \gamma_3$ as wanted. □

As a corollary of the previous lemma we obtain a fundamental property of `utcc`, i.e., *determinism*.

Theorem 3.7.1 (Determinism)**.** *Let α, β and β' be sequences of constraints. If both (α, β), $(\alpha, \beta') \in io(P)$ then for all $i > 0$, $\beta(i) \equiv \beta'(i)$.*

Proof. Assume that $P \overset{(a,c)}{\Longrightarrow} Q$, $P \overset{(a,c')}{\Longrightarrow} Q'$ and let $\gamma_1 \equiv \langle P, a\rangle$, $\gamma_2 \equiv \langle P, a\rangle$. If $\gamma_1 \not\longrightarrow$ then trivially $\gamma_2 \not\longrightarrow$, $c \equiv c'$ and $Q \equiv Q'$. Now assume that $\gamma_1 \longrightarrow^* \gamma_1' \equiv \langle c, P_1\rangle \not\longrightarrow$ and $\gamma_2 \longrightarrow^* \gamma_2' \equiv \langle c', P_2\rangle \not\longrightarrow$. By repeated applications of Lemma 3.7.1 we conclude $\gamma_1' \equiv \gamma_2'$ and then, $c \equiv c'$ and $P_1 \equiv P_2$. We therefore have $Q \equiv Q'$. □

3.8 Infinite Internal Behavior

In `tcc`, processes are supposed to respond "instantaneously" to the environment when an input is provided. This is akin to the *Perfect Synchrony Hypothesis* in reactive systems [Berry 1992]: program combinators are determinate primitives that respond instantaneously to input signals. For this reason, a `tcc` process must reach its resting point (where no further evolution is possible) in a finite number of internal transitions.

The abstraction operator in `utcc` may induce an infinite sequence of internal transitions within a time interval thus never producing an observable transition. The sources of infinite behavior may include:

- **Abstraction Loops**: Take for example

$$R = (\mathbf{abs}\ x; \mathtt{out}_1(x))\,(\mathbf{local}\ z)\,\mathbf{tell}(\mathtt{out}_1(z)))$$

  Each time R gets a link on $\mathtt{out}_1$, it generates a new link z on $\mathtt{out}_1$ thus causing infinite internal behaviors. A similar problem involves several abstractions producing mutual recursive behaviors. This kind of looping problems can be avoided by requiring for each $(\mathbf{abs}\ x; c)\, P$ that P must be a **next** expression. This restriction, however, may also disallow behaviors which will not cause infinite internal computations as those in Example 3.6.1 and 3.6.2.

- **Infinitely Many Substitutions**: Another source of infinite internal behaviors involves the constraint system under consideration. Let $R = (\mathbf{abs}\ x; c)\, P$. If the current store d entails $c[t/x]$ for infinitely many t's, then R will have to produce $P[t/x]$ for each such t's. This kind of infinite internal behaviors can be avoided by allowing only guards c so that for any d the set $\{t \mid d \models_\Delta c[t/x]\}$ modulo logic equivalence is finite. This seems inconvenient for the modelling of cryptographic knowledge as typically is done in process calculi: The presence of some messages entails the presence of arbitrary compositions among them (see Chapter 8).

We then define the fragment of *well-terminated* processes as such processes that do not exhibit infinite internal behavior. Formally:

Definition 3.8.1 (Well-termination)**.** *The process P is said to be* well-terminated *if and only if for every α such that $\alpha(i) \neq \mathtt{false}$ for each i, there exists α' such as $(\alpha, \alpha') \in io(P)$.*

The set of well-terminated processes constitute a meaningful fragment of `utcc`. We shall show that they are enough, for instance, to encode Turing-powerful formalisms (see Chapter 6), to give a declarative account to a language for structured communication (see Chapter 9) and to model multimedia interaction systems (see Chapter 9).

In the next chapter we consider an alternative symbolic operational semantics which deals with the above-mentioned internal termination problems. This semantics will allow us to describe the behavior of non well-terminated processes such as those arising from the verification of security protocols that we illustrate in Chapter 8.

3.9 Summary and Related Work

This chapter introduced the syntax and the operational semantics of `utcc`. We defined the input-output behavior of a process as well as its default output behavior. We illustrated how the interplay between abstractions and local processes allows for a name passing discipline in `utcc`. We also proved that the calculus is deterministic. Finally, we pointed out that there exist processes that exhibit infinitely many internal reductions and then, they do not produce any output. We shall deal with this problem in the next chapter.

The material of this chapter was originally published as [Olarte 2008c].

Related Work. In the CCP model, it is also possible to specify mobile behavior using logical variables to represent channels and unification to bind messages to channels [Saraswat 1993]. Recall that a logical variable can be bound to a value only once. Therefore, if two messages are sent through the same channel, they must be equal to avoid an inconsistent store. This problem is solved in [Laneve 1992] by considering *atomic* tells where the constraint c in $\mathbf{tell}(c)$ is added to the store d if the conjunction $c \wedge d$ is consistent. Channels are represented as "imperative style" variables by binding them to *streams* recording the current and the previous values. Therefore, a protocol is required since messages must compete for a position in such a stream. Notice that unlike CCP, Atomic CCP is non-deterministic. Take for example, $\mathbf{tell}(c).P \parallel \mathbf{tell}(\neg c)$. The execution of P depends on whether $\mathbf{tell}(\neg c)$ is scheduled for execution first or not. Furthermore, to the best of our knowledge, no logic characterizations have been given to this calculus.

In our approach, communication of messages is represented by predicates (i.e. constraints) of the form $\mathtt{out}(x)$, where `out` stands for a public (global) channel as in Example 3.6.1. Channel names can be explicitly modeled by using binary predicates of the form

$\texttt{out}(x, y)$ where y is the datum sent through the channel x as in Example 3.6.2. Therefore, it is possible to send different messages through the same channel.

In [Hildebrandt 2009], the authors show that modeling communication channels using binary predicates in `utcc` allows agents to guess channel names by using abstractions. For example, the process $(\mathbf{abs}\ x, y; \texttt{out}(x, y))\ P$ is able to capture all possible messages in transit due to the quantification of the channel name x. Then, a type system for constraints used as patterns in abstractions is proposed. Roughly speaking, the type system rules out processes where the variable representing the channel name is bound by an abstraction operator.

As it was pointed out in [Fages 2001], asks in CCP are not parametric in the sense that free-variables in the guard are not supposed to be universally quantified. For instance, the configuration $\langle \mathbf{when}\ c(x)\ \mathbf{do}\ P, c(0)\rangle$ does not have any internal transition since $c(0) \not\models_\Delta c(x)$. For certain constraint systems, it is possible to have the same effect of the universal quantification by using the interplay of asks and tells with the local operator. For example, assuming the Herbrand constraint system [Saraswat 1993], let $Q = \mathbf{when}\ \forall_{y,z}(x = [y|z])\ \mathbf{do}\ P$ be a process that splits the list x into $[y|z]$ and then executes P. We can define $Q' = \mathbf{when}\ \exists_{y,z}(x = [y|z])\ \mathbf{do}\ (\mathbf{local}\, y, z)\, (\mathbf{tell}(x = [y|z]) \parallel P)$ where $\mathbf{tell}(x = [y|z])$ unifies y and z to be respectively the head and the tail of the list x as expected. Nevertheless, this programming technique cannot be generalized to arbitrary constraint systems.

In Linear CCP [Fages 2001], the universal quantification in ask processes is explicit. Then a parametric ask $A(x)$ can be viewed as a process $\mathbf{when}\ c\ \mathbf{do}\ P$ with a variable x declared as a formal parameter much like the abstraction process defined here. Unlike standard CCP, asks in Linear CCP are not persistent. Then, $A(x)$ may evolve to either $P[y/x]$ or $P[z/x]$ if both $c[y/x]$ and $c[z/x]$ are entailed. Thus non-determinism arises. This kind of non-determinism can be avoided by using *persistent* parametric asks (with replication "!"). Forcing every ask to be persistent, however, makes the extension not suitable for modelling typical scenarios where a process stops after performing its query.

The abstraction operator in `utcc` can be then seen as a temporary persistent extension of the parametric ask in Linear CCP. Recall that $Q = (\mathbf{abs}\ \vec{x}; c)\, P$ executes $P[\vec{t}/\vec{x}]$ in the current time interval for all the sequences of terms $\vec{t}$ s.t $c[\vec{t}/\vec{x}]$ is entailed by the current store but Q evolves into **skip** after the end of the time unit.

Another difference between `utcc` and Linear CCP is the logic characterization these languages enjoy. In `utcc`, processes can be related to formulae in first-order linear-time temporal logic (see Chapter 5) while processes in Linear CCP correspond to formulae in Girard's Linear logic [Girard 1987]. Moreover, no closure operator semantics in the lines of standard CCP has been given to Linear CCP (we shall give a closure operator semantics to `utcc` in Chapter 7).

In [Buscemi 2007], the cc-pi calculus is proposed. This language results from the combination of the CCP model with a name-passing calculi. More precisely, cc-pi extends CCP by adding synchronous communication and by providing a treatment of names in terms of restriction and structural axioms closer to nominal calculi than to variables with existential quantification.

The cc-pi name passing discipline is reminiscent to that in the pi-F calculus [Wischik 2005] whose synchronization mechanism is global and, instead of binding formal names to actual names, it yields explicit fusions, i.e., simple constraints expressing name equalities. For example, the process $\bar{x}\langle y\rangle.P$ (sending y on x) synchronizes with $x\langle z\rangle.Q$ and evolves into $P \parallel Q \parallel y = z$. This approach differs from ours since mobility in cc-pi is achieved by means of the constructs inherited from pi-F.

The π^+-calculus [Díaz 1998] is an extension of the π-calculus with constraint agents that can perform tell and ask actions. Similarly as in cc-pi, the mobility of π^+comes from the

operands inherited from the π-calculus. Furthermore, no logic characterization has been given to π^+.

The language *LMNtal* [Ueda 2006] uses logical variables to specify mobile behavior as basic CCP does. Since LMNtal was designed as a unifying model of concurrency, it is non-deterministic. Furthermore, to our knowledge no correspondence between this languages and logic has been given.

All in all, the novelty of `utcc` is to allow for mobile behavior in the CCP model but preserving the appealing features of the initial language. Namely, (1) *determinism*, leading to rather simple and elegant denotational semantics based on solution of equations, and (2) *declarative view* of processes as formulae in logic.

Chapter 4

Symbolic Semantics

In the previous chapter we gave `utcc` an *operational semantics* and we showed how the abstraction operator may pose some technical problems. Namely, `utcc` processes may generate infinitely many substitutions thus causing divergent (i.e., infinite) internal computations. Since the observable relation in `utcc` is obtained from a finite number of internal reductions, it is not then possible to observe the behavior of non well-terminated processes.

In this chapter we address this problem by endowing `utcc` with a novel *symbolic semantics* that uses temporal constraints to represent finitely a possible infinite number substitutions. We shall show that without appealing to any syntactic restrictions like those in Section 3.8, this semantics guarantees that every sequence of internal transitions is finite. Furthermore, for the fragment of well-terminated processes, we prove that the outputs of both semantics correspond to each other.

To our knowledge, this is the first symbolic semantics in concurrency theory using temporal constraints as finite representations of substitutions.

4.1 Symbolic Intuitions

Before defining the symbolic semantics let us give some intuitions of its basic principles.

(A) **Substitutions as Constraints.** Take $R = (\mathbf{abs}\ x; c)\, P$. The operational semantics in Table 3.1 performs $P[t/x]$ for every t s.t $c[t/x]$ is entailed by the store d. Instead, the symbolic semantics we propose here dictates that R should produce $e = (d \wedge \forall x(c \Rightarrow d'))$ where, similarly, d' should be produced according to the symbolic semantics by P. Let t be an arbitrary term s.t $d \models_\Delta c[t/x]$. The idea is that if e' is operationally produced by $P[t/x]$ then e' should be entailed by $d'[t/x]$. Since $d \models_\Delta c[t/x]$ then $e \models_\Delta d'[t/x] \models_\Delta e'$. Therefore e entails the constraint that any arbitrary $P[t/x]$ produces.

(B) **Timed Dependencies in Substitutions.** The symbolic semantics represents as temporal constraints dependencies between substitutions from one time interval to another. For instance, suppose that for the process R above, $P = \mathbf{next}\,\mathbf{tell}(e)$. Operationally, once we move to the next time unit, the constraints produced are of the form $e[t/x]$ for those terms t's such that the final store d in the *previous* time unit entails $c[t/x]$. The symbolic semantics captures this behavior as $e' = (\ominus d) \wedge \forall x((\ominus c) \Rightarrow e)$ where $\ominus$ is the "previous" modality of first-order linear-time temporal logic (FLTL) [Manna 1991]. Intuitively, $\ominus c'$ means that c' holds in the previous time interval. This way, the information of the previous time units is transferred to the current one as *past* information to deal with next-guarded constructs in the body of abstractions.

In the sequel we formalize the idea of temporal formulae as constraints and we define the internal and observable reductions for the symbolic semantics. We then prove the correspondence of this semantics with respect to the operational semantics.

4.1.1 Future-Free Temporal Formulae as Constraints

As explained above, the symbolic semantics of `utcc` requires the past modality of FLTL to represent timed dependencies in substitutions. Let us then define the following fragment of the FLTL described in Section 2.4.

Definition 4.1.1 (Future-Free Formulae)**.** *A temporal formula is said to be* future-free *iff it does not contain occurrences of the modalities* $\Box$ *and* $\circ$.

We shall use FF to denote the set of future-free formulae with typical elements e, e'. *Sequences of future-free formulae are ranged by* $w, w', v, v', u, u', \ldots$.

From now on, given a constraint system with underlying first-order languages $\mathcal{L}$, we shall assume that constraints are built from $\mathcal{L}$ and the past modality $\ominus$. More precisely,

Definition 4.1.2 (FLTL Theories)**.** *Given a constraint system* (Δ, Σ) *with first-order language* $\mathcal{L}$*, the FLTL theory induced by* Δ*,* $T(\Delta)$ *is the set of FLTL sentences that are valid in all the* $\mathcal{L}$*-structures (or* $\mathcal{L}$*-models) of* Δ. *We write* $F \models_{T(\Delta)} G$ *iff* $(F \Rightarrow G) \in T(\Delta)$. *We omit "*$(\Delta)$*" in* $\models_{T(\Delta)}$ *when* $\Delta = \emptyset$.

We shall assume that processes and configurations are extended to include future-free formulae rather than just constraints. So, for example a process-store configuration of the form $\langle (\mathbf{abs}\ y; \ominus c)\, P, \ominus d \rangle$ may appear in the transitions of the symbolic semantics. We also assume that no process, other than the ones generated from the symbolic internal and observable transitions, can contain temporal modalities.

4.2 Symbolic Reductions

The internal and observable *symbolic* transitions $\longrightarrow_s, \Longrightarrow_s$ are defined as in Table 3.1 for the operational semantics with $\models_\Delta$ replaced with $\models_{T(\Delta)}$ (entailment of temporal formulae) and with the rules $\mathrm{R}_{\mathrm{ABS}}$ and $\mathrm{R}_{\mathrm{OBS}}$ replaced with $\mathrm{R}_{\mathrm{ABS-SYM}}$ and $\mathrm{R}_{\mathrm{OBS-SYM}}$ as in Table 4.1 respectively.

The rule $\mathrm{R}_{\mathrm{ABS-SYM}}$ represents with the temporal constraint $\forall \vec{x}(e \Rightarrow e')$ the substitutions that its operational counterpart $\mathrm{R}_{\mathrm{ABS}}$ would induce, as intuitively explained in Section 4.1 (A). Notice that in the reduction of P the variables $\vec{x}$ in e are hidden, via existential quantification, to avoid clashes with those in P.

The *future* function F_s in $\mathrm{R}_{\mathrm{OBS-SYM}}$ is similar to its operational counterpart F in Definition 3.4.2. However, F_s records the final global and local stores as well as abstraction guards as past information. As explained in Section 4.1 (B), this past information is needed in the next time unit when next guarded processes occur in the body of an abstraction.

Definition 4.2.1. *Let* F_s *be a partial function defined by* $F_s(P, e) = \mathbf{tell}(\ominus e) \parallel F'_s(P)$ *where:*

$$F'_s(P) = \begin{cases} \mathbf{skip} & \text{if } P = \mathbf{skip} \\ (\mathbf{abs}\ \vec{x}; \ominus e)\, F'_s(Q) & \text{if } P = (\mathbf{abs}\ \vec{x}; e)\, Q \\ F'_s(P_1) \parallel F'_s(P_2) & \text{if } P = P_1 \parallel P_2 \\ (\mathbf{local}\, \vec{x}; \ominus e)\, F'_s(Q) & \text{if } P = (\mathbf{local}\, \vec{x})\, (Q \parallel \mathbf{tell}(\ominus(e))) \\ Q & \text{if } P = \mathbf{next}\, Q \\ Q & \text{if } P = \mathbf{unless}\ c\ \mathbf{next}\, Q \end{cases}$$

Let us introduce some notation about (sequences of) temporal formulae that we shall use in the sequel.

$$\mathrm{R}_{\mathrm{ABS-SYM}}\ \frac{\langle P, \exists \vec{x} e\rangle \longrightarrow_s \langle Q, e'' \wedge \exists \vec{x} e\rangle}{\langle (\mathbf{abs}\ \vec{x}; e')\, P, e\rangle \longrightarrow_s \langle (\mathbf{abs}\ \vec{x}; e')\, Q, e \wedge \forall \vec{x}(e' \Rightarrow e'')\rangle}$$

$$\mathrm{R}_{\mathrm{OBS-SYM}}\ \frac{\langle P, e\rangle \longrightarrow_s^* \langle Q, e'\rangle \not\longrightarrow_s}{P \xRightarrow{(e,e')}_s F_s(Q, e')}$$

Table 4.1: Symbolic Rules for Internal and Observable Transitions. The function F_s is given in Definition 4.2.1.

Notation 4.2.1. *Let e and e' be future-free formulae. We write $e \succeq e'$ whenever $e \models_{T(\Delta)} e'$. If $e \succeq e'$ and $e' \succeq e$ we write $e \equiv e'$. If $e \succeq e'$ and $e \not\equiv e'$ then we write $e \succ e'$. We extend $\succeq$, $\succ$ and $\equiv$ to sequences of future-free formulae: $w \succeq v$ (resp. $w \equiv v$) iff for all $i > 0$, $w(i) \succeq v(i)$ (resp. $w(i) \equiv v(i)$). We shall write $w \succ v$ if $w \succeq v$ and there exists $i > 0$ s.t. $w(i) \succ v(i)$.*

If $P = P_1 \xRightarrow{(e_1,e_1')}_s P_2 \xRightarrow{(e_2,e_2')}_s \ldots$, we write $P \xRightarrow{(w,w')}_s$ if $w = e_1.e_2\ldots$ and $w' = e_1'.e_2'\ldots$. We shall write $P \sim_s^{io} Q$ whenever for any sequence w, $P \xRightarrow{(w,v)}_s$ iff $Q \xRightarrow{(w,v)}_s$.

Finally, similar to the operational semantics, we shall say that P eventually outputs c, notation $P \Downarrow_s^c$, if $P \xRightarrow{(\mathtt{true}^\omega, w)}_s$ and there exists $i > 0$ s.t. $w(i) \models_{T(\Delta)} c$.

4.2.1 The Abstracted-Unless Free Fragment

It is worth noticing that the symbolic semantics fails to give a representation of **unless** processes in the scope of an abstraction. Basically, the problem is to represent negation of entailment as a logical formula. Let us explain this with an example. Take $P = (\mathbf{abs}\ x; \mathtt{true})\, Q$ and let $Q = \mathbf{unless}\ e\ \mathbf{next}\,\mathbf{tell}(e')$. Assume that the final store in the first time unit when running P is d. Operationally, $\mathbf{tell}(e')[t/x]$ is executed in the second time unit for those t's such that $d \not\models_\Delta e[t/x]$. Following Section 4.1, one may try to capture this in the symbolic semantics with the temporal constraint $\ominus d \wedge \forall x(\ominus(\neg e) \Rightarrow e')$ as if we had $Q = Q' = \mathbf{when}\ \neg e\ \mathbf{do}\ \mathbf{next}\,\mathbf{tell}(e')$. Nevertheless, this wrongly assumes that Q and Q' behave the same (see Remark 3.2.1).

Taking the previous observation into account, we define the abstracted-unless free fragment of `utcc` processes.

Definition 4.2.2 (Abstracted-unless free Processes)**.** *We say that P is* abstracted-unless free *if there is no processes of the form* **unless** c **next** Q *in P under the scope of an abstraction.*

The following proposition introduces an obvious fact on this fragment.

Proposition 4.2.1 (Abstracted-unless freeness Invariance)**.** *Let P be an abstracted-unless free process. If $P \xRightarrow{(e,d)}_s Q$ then Q is also abstracted-unless free.*

Proof. Given an abstracted-unless free process P, one can easily show that if $\langle P, e\rangle \longrightarrow_s \langle P', e'\rangle$ then P' is also abstracted-unless free. One concludes by noticing that for any P abstracted-unless free and future-free formula e, $F_s(P, e)$ is also abstracted-unless free. □

Abstract-unless free processes represent a meaningful and practical fragment of `utcc`. For example, this fragment allows us to model Security Protocols as we shall show in Chapter 8. Then, we can use the symbolic semantics to observe the behavior of the processes modeling those protocols which in general are non well-terminated.

4.2.2 Past-Monotonic Sequences

As explained above, the future function in Definition 4.2.1 transfers the final store of a time unit to the next one as a past formula. Therefore, for any process P, if $P \xrightarrow{(w,w')} \cdots$ then w' is a past-monotonic sequence in the following sense.

Definition 4.2.3 (Past-Monotonic Sequences, PM)**.** *We say that an infinite sequence of future-free formulae w is* past-monotonic *iff for all $i > 1$, $w(i) \models_{T(\Delta)} \ominus w(i-1)$. The set of infinite sequences of past-monotonic formulae is denoted by PM.*

Given a sequence of future free formulae, we can add to it the corresponding past information to obtain a past-monotonic sequence as follows.

Notation 4.2.2. *Given a sequence $e_1.e_2...$, we shall use $\overline{e_1.e_2...}$ to denote the* past-monotonic *sequence*

$$e_1.(e_2 \wedge \ominus e_1).(e_3 \wedge \ominus e_2 \wedge \ominus^2 e_1)...$$

Notice that if v is a past-monotonic sequence then $v \equiv \overline{v}$.

4.3 Properties of the Symbolic Semantics

In this section we state some properties of the symbolic semantics. Among them, we shall prove that similar to the operational semantics, the symbolic semantics is confluent. Furthermore, for all process and input, every sequence of symbolic internal transitions is finite. This means that the symbolic semantics solves the infinite internal behavior problem of the operational semantics when considering non well-terminated processes. Moreover, a remarkable property of the symbolic semantics is that the symbolic output of a process is in some sense "insensitive" to the input. More precisely, we shall show that the contribution of a process to the output is the same regardless the input.

Determinism. We start by showing that, similar to the operational semantics, the symbolic internal transitions are *confluent.*

Lemma 4.3.1 (Confluence –Symbolic Semantics–)**.** *Suppose that $\gamma_0 \longrightarrow_s \gamma_1$, $\gamma_0 \longrightarrow_s \gamma_2$ and $\gamma_1 \not\equiv \gamma_2$. Then, there exists γ_3 such that $\gamma_1 \longrightarrow_s \gamma_3$ and $\gamma_2 \longrightarrow_s \gamma_3$.*

Proof. Similarly to the proof of Lemma 3.7.1, we define the size of γ as $M(P)$ (see Definition 3.4.3). We only consider the case for the abstraction operator. The other cases are the same as in Lemma 3.7.1. Thus let $\gamma_0 \equiv \langle P_0, c_0\rangle$ with $P_0 = (\mathbf{abs}\ \vec{x}; c)\, Q$. One can verify that $\gamma_1 \equiv \langle P_1, c_1\rangle$ where P_1 takes the form $(\mathbf{abs}\ \vec{x}; c)\, Q_1$ and $\gamma_2 \equiv \langle P_2, c_2\rangle$ where P_2 takes the form $(\mathbf{abs}\ \vec{x}; c)\, Q_2$. From the assumption $\gamma_1 \not\equiv \gamma_2$, it must be the case that $Q_1 \not\equiv Q_2$. Then, by induction there exists γ_3' such that $\gamma_0' \equiv \langle Q, \exists_{\vec{x}}(c_0)\rangle \longrightarrow_s \langle Q_1, \exists_{\vec{x}}(c_0) \wedge c_1'\rangle \equiv \gamma_1'$, $\gamma_0' \longrightarrow_s \langle Q_2, \exists_{\vec{x}}(c_0) \wedge c_2'\rangle \equiv \gamma_2'$ and γ_1', γ_2' commute to $\gamma_3' \equiv \langle Q_3', c_3'\rangle$. By the rule $\mathrm{R}_{\mathrm{ABS-SYM}}$ we have that γ_1 and γ_2 commute to $\gamma_3 \equiv \langle P_3, c_3\rangle$ where $P_3 = (\mathbf{abs}\ \vec{x}; c)\, Q_3'$ and $c_3 = c_0 \wedge \forall_{\vec{x}}(c \Rightarrow c_3')$ as wanted. □

As a corollary of the previous lemma we have that the symbolic outputs of a process are equivalent up to logical equivalence.

Theorem 4.3.1 (Determinism). *Let w be a sequence of constraints, w' and v be sequences of past-monotonic formulae and P be an abstracted-unless free process. If $P \xrightarrow{(w,v)}_s$ and $P \xrightarrow{(w,v')}_s$, then for all $i > 0$, $v(i) \equiv v'(i)$.*

Proof. Similar to the proof of Determinism in the operational semantics (Theorem 3.7.1). □

Finite Number of Symbolic Internal Reductions. One of the most important properties of the symbolic semantics is that it solves the problem of infinitely many internal reductions described in Section 3.8. More precisely,

Lemma 4.3.2 (Finiteness of the Symbolic Internal Reductions). *Given a configuration $\gamma_0 = \langle P, e\rangle$, there exist configurations $\gamma_1, ..., \gamma_n$ with $n < \omega$ s.t.*

$$\gamma_0 \longrightarrow_s \gamma_1 \longrightarrow_s \gamma_2 \longrightarrow_s \cdots \longrightarrow_s \gamma_n \not\longrightarrow_s$$

Proof. Observe that next-guarded processes do not exhibit any internal transition. Then, define $M'(P)$ as $M(P)$ in Definition 3.4.3 but let $M'(\mathbf{next}\, P_1) = 0$. By induction on the size of P, one can show that if $\langle P, e\rangle \longrightarrow \langle P', e'\rangle$ then there exists $P'' \equiv P'$ such that $M'(P) > M'(P'')$. Therefore, the number of symbolic internal reductions of P is bound by $M'(P)$. □

From the previous lemma we straightforwardly deduce the following corollary.

Corollary 4.3.1 (Finite Symbolic Internal Transitions). *Given an abstracted-unless free process P, for any sequence of future free formulae w there exists w' such that $P \xrightarrow{(w,w')}_s$.*

Non-blocking Symbolic Abstractions. In addition to the fact that the symbolic semantics does not exhibit infinite internal behavior, there is another fundamental difference between both semantics. The rule for the abstraction in the symbolic semantics *does not depend* on the current store, i.e., an abstraction in the symbolic semantics *does not block* until the entailment of its guard. Take for example $P = (\mathbf{abs}\ x; c)\,\mathbf{tell}(d)$ and assume the configuration $\gamma_1 = \langle P, c'\rangle$ where there is no t such that $c' \models_\Delta c[t/x]$. Operationally, there is no an internal reduction, i.e., $\gamma_1 \not\longrightarrow$. Nevertheless, in the symbolic semantics we have a derivation of the form

$$\gamma_1 \longrightarrow_s \langle \mathbf{skip}, c' \wedge \forall_x(c \Rightarrow d)\rangle \not\longrightarrow_s$$

Later on we shall show that the final store c' in the operational semantics and $d' = c' \wedge \forall_x(c \Rightarrow d)$ in the symbolic one are related. Roughly speaking, c' and d' entail the same basic constraints (see Definition 3.1.2).

4.3.1 Normal Form of Processes

The following definition introduces a normal form of processes useful for proving some of the results in this dissertation.

Definition 4.3.1 (Normal Forms). *We say that the* `utcc` *process P is in* normal *form if it takes the form*

$$P \equiv (\mathbf{local}\,\vec{x}; c)\left(\begin{array}{l} \prod_{i\in I} \mathbf{tell}(c_i) \ \parallel \prod_{j\in J} (\mathbf{abs}\ \vec{y}_j; c_j)\, P_j \parallel \prod_{k\in K} \mathbf{next}\, P_k \parallel \\ \prod_{l\in L} \mathbf{unless}\ c_l\ \mathbf{next}\, P_l \parallel \prod_{m\in M} !\, P_m \end{array}\right)$$

where each P_j, P_k, P_l and P_m are themselves in normal form. We assume the variables in $\vec{x}$ and $\vec{y_j}$ do not appear bound elsewhere, i.e., $\vec{x}$, the $\vec{y}'s$ and the bound names of the P_k's, P_l's and P_m's are pairwise distinct. Furthermore, no variable appears both free and bound in the process P.

The proposition below states that for all process P there exists P' structurally congruent to P in normal form.

Proposition 4.3.1. *Given a process P, there exists P' in the normal form of Definition 4.3.1 such that $P \equiv P'$.*

Proof. The proof proceeds trivially by induction on the structure of P. □

The following observation points out that for a configuration $\gamma = \langle P, e\rangle$ such that there is no symbolic internal transitions from γ, the process P takes a simpler normal form.

Observation 4.3.1. *Let P be a process in normal form and $\gamma = \langle P, e\rangle$. If $\gamma \not\longrightarrow_s$ then P must take the form*

$$P \equiv \quad (\mathbf{local}\,\vec{x}; c)\left(\prod_{j\in J}(\mathbf{abs}\ \vec{x}_j; c_j)\,P_j \parallel \prod_{k\in K}\mathbf{next}\,P_k \parallel \prod_{l\in L}\mathbf{unless}\ c_l\ \mathbf{next}\,P_l\right)$$

and for all $l \in L$, $c \wedge \exists_{\vec{x}} e \not\models_{T(\Delta)} c_l$. Furthermore, for all $j \in J$, $\langle P_j, c \wedge \exists_{\vec{x}} e\rangle \not\longrightarrow_s$.

The previous observation follows from the fact that if a constraint c_l can be entailed from e, then there is a symbolic transition from γ where the process **unless** c_l **next** P_l evolves into **skip**. Similarly, if a process P_j may evolve, then there exists a transition from γ using the rule $\mathrm{R}_{\mathrm{ABS-SYM}}$.

4.3.2 Symbolic Output Invariance

In this section we prove that the symbolic output of a process in a time unit is independent of the input, i.e., the contribution of a process to the final output is the same regardless the input from the environment. This can intuitively be explained from the fact that only the symbolic rule for $P =$ **unless** c **next** Q depends on the current store and P can only add information to the store in the next time unit.

Firstly, we prove that the symbolic output of a process P is independent from the context running in parallel with P.

Lemma 4.3.3 (Parallel Composition Invariance)**.** *Let P and Q be abstracted-unless free* `utcc` *processes and c be a constraint. If $P \overset{(c,d)}{\Longrightarrow}_s P'$ and $Q \overset{(c,e)}{\Longrightarrow}_s Q'$, then there exist P'' and Q'' s.t $P \parallel Q \overset{(c,d\wedge e)}{\Longrightarrow}_s P'' \parallel Q''$.*

Proof. Assume the following derivations of $P = P_1$ and $Q = Q_1$ with $c = d_1 = e_1$

$$\langle P_1, d_1\rangle \longrightarrow_s \langle P_2, d_2\rangle \longrightarrow_s^* \langle P_n, d_n\rangle \not\longrightarrow_s$$
$$\langle Q_1, e_1\rangle \longrightarrow_s \langle Q_2, e_2\rangle \longrightarrow_s^* \langle Q_m, e_m\rangle \not\longrightarrow_s$$

By Proposition 4.3.1 and Observation 4.3.1 we must have that

$$P_n \equiv \quad (\mathbf{local}\,\vec{x}; c_1)\left(\prod_{j\in J_1}(\mathbf{abs}\ \vec{x}_j; c_j)\,P_j \parallel \prod_{k\in K_1}\mathbf{next}\,P_k \parallel \prod_{l\in L_1}\mathbf{unless}\ c_l\ \mathbf{next}\,P_l\right)$$
$$Q_m \equiv \quad (\mathbf{local}\,\vec{x}; c_1')\left(\prod_{j\in J_2}(\mathbf{abs}\ \vec{x}_j; c_j)\,Q_j \parallel \prod_{k\in K_2}\mathbf{next}\,Q_k \parallel \prod_{l\in L_2}\mathbf{unless}\ c_l\ \mathbf{next}\,Q_l\right)$$

where 1) for all $j \in J_1$, $\langle P_j, d_n \wedge \exists_{\vec{x}}(c_1)\rangle \not\longrightarrow_s$; 2) for all $j \in J_2$ $\langle Q_j, e_m \wedge \exists_{\vec{x}}(c'_1)\rangle \not\longrightarrow_s$ and 3) for all $l \in L_1$, $l' \in L_2$, $c \wedge \exists_{\vec{x}} d_n \not\models_{T(\Delta)} c_l$ and $c \wedge \exists_{\vec{x}} e_m \not\models_{T(\Delta)} c_{l'}$.

We can show that there exists a derivation of the form

$$\begin{array}{lll} \langle P_1 \parallel Q_1, c\rangle \longrightarrow_s^* & \langle P_n \parallel Q_1, d_n\rangle \longrightarrow_s^* & \\ & \langle P_n \parallel Q_m, d_n \wedge e_m\rangle \longrightarrow_s^* & \\ & \langle P'_n \parallel Q'_m, d_n \wedge e_m\rangle \not\longrightarrow_s & \end{array}$$

where the derivations in $\langle P_n, d_n \wedge e_n\rangle \longrightarrow^* \langle P'_n, d_n \wedge e_n\rangle$ can only use the rules $\mathrm{R_{STR}}$ and $\mathrm{R_{UNL}}$ (i.e., some of the **unless** processes in P_n evolved into **skip**). Similarly for the evolution from Q_n into Q'_n. Therefore, there exists $L'_1 \subseteq L_1$ and $L'_2 \subseteq L_2$ s.t.

$$\begin{array}{lll} P'_n \equiv & (\mathbf{local}\,\vec{x}; c_1) & \left(\prod_{j\in J_1} (\mathbf{abs}\ \vec{x}_j; c_j)\, P_j \parallel \prod_{k\in K_1} \mathbf{next}\, P_k \parallel \prod_{l\in L'_1} \mathbf{unless}\ c_l\ \mathbf{next}\, P_l\right) \\ Q'_m \equiv & (\mathbf{local}\,\vec{x}; c'_1) & \left(\prod_{j\in J_2} (\mathbf{abs}\ \vec{x}_j; c_j)\, Q_j \parallel \prod_{k\in K_2} \mathbf{next}\, Q_k \parallel \prod_{l\in L'_2} \mathbf{unless}\ c_l\ \mathbf{next}\, Q_l\right) \end{array}$$

Since $d = d_n$ and $e = e_m$ we conclude $P \parallel Q \xrightarrow{(c,d\wedge e)}_s F_s(P'_n, d) \parallel F_s(Q'_m, e)$. □

The previous proof can be straightforwardly adapted to show that the contribution of a process to the final output is the same regardless the input from the environment.

Lemma 4.3.4 (Input Invariance)**.** *Let P, Q be abstracted-unless free processes such that $P \xrightarrow{(e,e\wedge d)}_s Q \parallel \mathbf{tell}(\ominus(e \wedge d))$. For all future-free formula e' there exists Q' s.t.*

$$P \xrightarrow{(e',e'\wedge d)}_s Q' \parallel \mathbf{tell}(\ominus(e' \wedge d))$$

Furthermore, $Q \equiv Q'$ if for all basic constraint c, $e' \wedge d \models_{T(\Delta)} c$ iff $e \wedge d \models_{T(\Delta)} c$.

Proof. Directly from the proof of Lemma 4.3.3. Notice that the processes Q and Q' may differ only in that some **unless** processes in P evolve into **skip** under input e and not under input e' or vice versa. If both $e \wedge d$ and $e' \wedge d$ entail the same basic constraints, then trivially $Q \equiv Q'$. □

4.3.3 The Monotonic Fragment

Note that, unlike the other constructs in `utcc`, the **unless** operator exhibits non-monotonic input-output behavior in the following sense: Given $w' \succeq w$ and $P = \mathbf{unless}\ c\ \mathbf{next}\, Q$, if $P \xrightarrow{(w,v)}_s$ and $P \xrightarrow{(w',v')}_s$, then it may be the case that $v' \not\succeq v$. For example, take $Q = \mathbf{tell}(d)$, $w = \mathtt{true}^\omega$ and $w' = c.\mathtt{true}^\omega$. In this case, $v = \overline{\mathtt{true}.d.\mathtt{true}^\omega}$, $v' = \overline{\mathtt{true}^\omega}$, $w' \succeq w$ but $v' \not\succeq v$.

We then define the monotonic fragment of `utcc` processes as follows.

Definition 4.3.2 (Monotonic Processes)**.** *We say that P is a* monotonic process *iff P does not have occurrences of processes of the form* **unless** c **next** Q.

The following proposition introduces an obvious fact on this fragment.

Proposition 4.3.2 (Monotonic Invariance)**.** *Let P be a monotonic process. If $P \xrightarrow{(e,d)}_s Q$ then Q is also monotonic.*

Proof. Immediate. □

For the monotonic fragment of `utcc`, Lemma 4.3.4 can be strengthen in two ways. On the one hand, the Lemma 4.3.5 states that the process Q to be executed in the next time unit is the same regardless the input of the process P. On the other hand, Lemma 4.3.6 generalizes this result to infinite sequences of observable transitions.

Lemma 4.3.5 (Monotonic Input Invariance)**.** *Let P, Q be monotonic processes and d be a constraint such that $P \xrightarrow{(d,d\wedge e)}_s Q \parallel \mathbf{tell}(\ominus(d \wedge e))$. For all future-free formula d',*

$$P \xrightarrow{(d',d'\wedge e)}_s Q \parallel \mathbf{tell}(\ominus(d' \wedge e))$$

Proof. Directly from the proof of Lemma 4.3.4 and using the fact that P does not have occurrences of **unless** processes. □

Lemma 4.3.6 (Monotonic Input Insensitiveness)**.** *Let P be a monotonic process and u, v, w be past-monotonic sequences. If $P \xrightarrow{(w,w\wedge v)}_s$ then $P \xrightarrow{(u,u\wedge v)}_s$.*

Proof. Immediate by repeated applications of the Lemma 4.3.5. □

For the monotonic fragment it is also possible to relate the behavior of the process P and the behavior of $P[\vec{t}/\vec{x}]$. This is central to prove the semantic correspondence of the symbolic and the operational semantics when considering the case of the abstraction operator. Before doing this, we need the Lemma 4.3.7 that shows that $P[\vec{t}/\vec{x}]$ and $(\mathbf{local}\,\vec{x})\,(P \parallel !\,\mathbf{tell}(\vec{x} = \vec{t}))$ behave the same much like in logic $F[\vec{t}/\vec{x}] \equiv \exists_{\vec{x}}(F \wedge \Box\vec{x} = \vec{t})$.

Recall that the substitution $[\vec{t}/\vec{x}]$ is said to be admissible if the variables in $\vec{x}$ do not appear in $\vec{t}$, i.e., $adm(\vec{x}, \vec{t})$ (see Convention 3.1.1). Recall also that $\sim^{io}$ is the input-output equivalence in Definition 3.7.1.

Lemma 4.3.7. *Let P be a `utcc` process and $\vec{x}$ be a sequence of pairwise distinct variables. Let $\vec{t} \in \mathcal{T}^{|\vec{x}|}$ and $[\vec{t}/\vec{x}]$ be an admissible substitution. We have the following*

1. $P[\vec{t}/\vec{x}] \sim^{io} (\mathbf{local}\,\vec{x})\,(P \parallel !\,\mathbf{tell}(\vec{x} = \vec{t}))$
2. $P[\vec{t}/\vec{x}] \sim^{io}_s (\mathbf{local}\,\vec{x})\,(P \parallel !\,\mathbf{tell}(\vec{x} = \vec{t}))$

Proof. We only prove (1). The proof of (2) is analogous but considering instead the symbolic reduction relation.

($\Rightarrow$) We shall prove that for any c, if $P[\vec{t}/\vec{x}] \xrightarrow{(c,c')} P'[\vec{t}/\vec{x}]$ then $(\mathbf{local}\,\vec{x})\,(P \parallel !\,\mathbf{tell}(\vec{x} = \vec{t})) \xrightarrow{(c,c')} (\mathbf{local}\,\vec{x})\,(P' \parallel !\,\mathbf{tell}(\vec{x} = \vec{t}))$. The conclusion follows from repeated applications of the following reasoning.

Assume by alpha conversion that $\vec{x} \notin fv(c)$. Notice that $\vec{x}$ does not occur free neither in $P[\vec{t}/\vec{x}]$ nor in $(\mathbf{local}\,\vec{x})\,(P \parallel !\,\mathbf{tell}(\vec{x} = \vec{t}))$. One can show that if $\langle P[\vec{t}/\vec{x}], c\rangle \longrightarrow^* \langle P'[\vec{t}/\vec{x}], c'\rangle \not\longrightarrow$ then it must be the case that $\langle P \parallel \mathbf{tell}(\vec{x} = \vec{t}), c\rangle \longrightarrow^* \langle P', c' \wedge \vec{x} = \vec{t}\rangle \not\longrightarrow$. Since $\vec{x} \notin fv(c)$, one can verify that

$$\langle (\mathbf{local}\,\vec{x})\,(P \parallel !\,\mathbf{tell}(\vec{x} = \vec{t})), c\rangle \longrightarrow^* \langle (\mathbf{local}\,x; c' \wedge \vec{x} = \vec{t})\,(P' \parallel \mathbf{next}\,!\,\mathbf{tell}(\vec{x} = \vec{t})), c''\rangle \not\longrightarrow$$

where $c'' = \exists_{\vec{x}}(c' \wedge \vec{x} = \vec{t}) = c'[\vec{t}/\vec{x}]$. From $\vec{x} \notin fv(c) \cup fv(P[\vec{t}/\vec{x}])$ and the fact that $\langle P[\vec{t}/\vec{x}], c\rangle \longrightarrow^* \langle P'[\vec{t}/\vec{x}], c'\rangle$ we derive that $\vec{x} \notin fv(c')$ and then, $c'' = c'$. By noticing that $F(Q[\vec{t}/\vec{x}]) \equiv F(Q)[\vec{t}/\vec{x}]$ we conclude $P[\vec{t}/\vec{x}] \xrightarrow{(c,c')} F(P'[\vec{t}/\vec{x}])$ and

$$(\mathbf{local}\,\vec{x})\,(P \parallel !\,\mathbf{tell}(\vec{x} = \vec{t})) \xrightarrow{(c,c')} (\mathbf{local}\,\vec{x})\,(F(P') \parallel !\,\mathbf{tell}(\vec{x} = \vec{t}))$$

($\Leftarrow$) The *only if* part can be obtained analogously by reversing the proof of the "if" case.

□

Now we are ready to relate the behavior of the processes P and $P[\vec{t}/\vec{x}]$.

Lemma 4.3.8. *Let P be a monotonic process and w, w' be past-monotonic sequences such that $\vec{x} \notin fv(w)$. If $P \xrightarrow{(w,w')}_s$ and $[\vec{t}/\vec{x}]$ is an admissible substitution, then $P[\vec{t}/\vec{x}] \xrightarrow{(w,w'[\vec{t}/\vec{x}])}_s$*

Proof. Assume that $[\vec{t}/\vec{x}]$ is an admissible substitution. Let $w = e_1.e_2.e_3...$, $w' = e'_1.e'_2.e'_3...$ and assume that $P \xrightarrow{(w,w')}_s$, i.e., there is a derivation of the form

$$P = P_1 \xrightarrow{(e_1,e'_1)}_s P_2 \xrightarrow{(e_2,e'_2)}_s P_3 \xrightarrow{(e_3,e'_3)}_s \dots$$

Let $v = \overline{(\vec{x} = \vec{t})^\omega} = d_1.d_2.d_3...$ (see Notation 4.2.2). By Lemma 4.3.6 there exist $P'_1, P'_2, P'_3, ...$ such that

$$P_1 = P'_1 \xrightarrow{(e_1 \wedge d_1, e'_1 \wedge d_1)}_s P'_2 \xrightarrow{(e_2 \wedge d_2, e'_2 \wedge d_2)}_s P'_3 \xrightarrow{(e_3 \wedge d_3, e'_3 \wedge d_3)}_s \dots$$

Let $Q = !\,\mathbf{tell}(\vec{x} = \vec{t})$. Since $v = \overline{(\vec{x} = \vec{t})^\omega}$, one can verify that

$$P'_1 \parallel Q \xrightarrow{(e_1, e'_1 \wedge d_1)}_s P'_2 \parallel Q \parallel \mathbf{tell}(\ominus(d_1)) \xrightarrow{(e_2, e'_2 \wedge d_2)}_s P'_3 \parallel Q \parallel \mathbf{tell}(\ominus(d_2)) \xrightarrow{(e_3, e'_3 \wedge d_3)}_s \dots$$

Let $w'' = e''_1, e''_2.e''_3...$ such that $e''_i = e'_i \wedge d_i$. Since w'' is past-monotonic we have

$$\begin{array}{lcl} e''_1 & = & e'_1 \wedge (\vec{x} = \vec{t}) \\ e''_2 & = & e'_2 \wedge (\vec{x} = \vec{t}) \wedge \ominus(\vec{x} = \vec{t}) \\ \dots & & \\ e''_n & = & e'_n \wedge (\vec{x} = \vec{t}) \bigwedge_{1 \le j \le n-1} \ominus^j(\vec{x} = \vec{t}) \end{array}$$

Since $\vec{x} \notin fv(w)$, we must have a derivation of the form

$$\begin{array}{lll} (\mathbf{local}\,\vec{x})\,(P'_1 \parallel Q) & \xrightarrow{(e_1, \exists_{\vec{x}} e''_1)}_s & (\mathbf{local}\,\vec{x}; \ominus e''_1)\,(P'_2 \parallel Q) \\ & \xrightarrow{(e_2, \exists_{\vec{x}} e''_2)}_s & (\mathbf{local}\,\vec{x}; \ominus e''_2)\,(P'_3 \parallel Q) \\ & \dots & \end{array}$$

From the fact that $\exists_{\vec{x}}(F \wedge \Box\vec{x} = \vec{t}) = F[\vec{t}/\vec{x}]$ for any formula F, $\exists_{\vec{x}} e''_1 = e'_1[\vec{t}/\vec{x}]$. By definition of the symbolic future function, $e'_i \models_{T(\Delta)} \ominus(e'_{i-1})$ for $i > 1$ and then

$$\begin{array}{lcl} \exists_{\vec{x}} e''_i & \equiv & \exists_{\vec{x}}(e'_i \wedge \vec{x} = \vec{t} \bigwedge_{1 \le j \le i-1} \ominus^j(\vec{x} = \vec{t})) \\ & \equiv & \exists_{\vec{x}}(e'_i \wedge \vec{x} = \vec{t} \wedge \ominus(e'_{i-1} \wedge \vec{x} = \vec{t}) \wedge \ominus^2(e'_{i-2} \wedge \vec{x} = \vec{t}) \wedge ... \wedge \ominus^{i-1}(e'_1 \wedge \vec{x} = \vec{t})) \\ & \equiv & (e'_i \wedge \ominus(e'_{i-1}) \wedge \dots \wedge \ominus^{i-1}(e'_1))[\vec{t}/\vec{x}] \equiv e'_i[\vec{t}/\vec{x}] \end{array}$$

By Lemma 4.3.7, $P[\vec{t}/\vec{x}] \sim^{io}_s (\mathbf{local}\,\vec{x})\,(P \parallel !\,\mathbf{tell}(\vec{x} = \vec{t}))$ and we conclude $P[\vec{t}/\vec{x}] \xrightarrow{(w,w'[\vec{t}/\vec{x}])}_s$.

□

4.4 Relating the SOS and the Symbolic Semantics

This section is devoted to proving the correspondence between the operational and the symbolic semantics. Recall that while the operational semantics outputs basic constraints, the symbolic semantics outputs future-free formulae. Then, we shall show that the basic

constraints entailed from the operational and the symbolic outputs of a process are the same.

We shall proceed as follows. We first show how the local variables $\vec{x}$ in a process of the form $(\mathbf{local}\,\vec{x}; c)\,Q$ can be replaced by "fresh variables". That is, variables that do not appear elsewhere in a process or the store. We do this in a similar way as existential quantifiers are eliminated in first-order formulae by Skolemization. Then, relaying in a previous result in CCP, we show that the output of P and that of the resulting process P' without local binders are the same when existentially quantifying on the fresh variables. Next, we give a simple characterization of the operational and the symbolic outputs of a process without local operators. Finally, appealing to this characterization and the lemmata in the previous section we establish the semantic correspondence in Theorem 4.4.1.

4.4.1 Elimination of local processes

In [Mendler 1995] and [Nielsen 2002b] it was shown that the semantics of the local operator can be redefined by making use of *fresh* variables in each transition. As in [Mendler 1995], we use the notion of *fresh* variable meaning that it does not occur elsewhere in a process or the store. This change in the semantics will allow us to get rid of the local operators, thus simplifying the proof of the correspondence between de SOS and the Symbolic semantics.

Assume that the set of variables $\mathcal{V}$ is partitioned into two infinite sets $\mathcal{F}$ and $\mathcal{V} - \mathcal{F}$. We shall assume that the fresh variables are taken from $\mathcal{F}$ and that no input from the environment or process, other than the ones generated when reducing a local process, can contain variables in $\mathcal{F}$. Following [Mendler 1995], one can redefine the rule for the **local** operator as follow:

$$\mathrm{R}'_{\mathrm{LOC}}\ \frac{\langle P[\vec{y}/\vec{x}], d \wedge c[\vec{y}/\vec{x}]\rangle \longrightarrow \langle P', d'\rangle \qquad \vec{y}\ \text{ is fresh}}{\langle (\mathbf{local}\,\vec{x}; c)\,P, d\rangle \longrightarrow \langle P', d'\rangle}$$

The fresh variables introduced by $\mathrm{R}'_{\mathrm{LOC}}$ are not to be visible from the outside. Therefore, we hide the variables in $\mathcal{F}$ by existential quantification. More precisely, we can replace the rule for the observable transitions $\mathrm{R}_{\mathrm{OBS}}$ by the rule

$$\mathrm{R}'_{\mathrm{OBS}}\ \frac{\langle P, c\rangle \longrightarrow^{*} \langle Q, d\rangle \not\longrightarrow}{P \overset{(c, \exists_{\mathcal{F}} d)}{\Longrightarrow} F(Q)}$$

In the sequel we provide a map from a process P to a process P' without occurrences of local operators such that P and P' are (symbolic) input-output equivalents. The idea is to replace $P = (\mathbf{local}\,\vec{x}; c)\,Q$ with $P' = (\mathbf{tell}(c) \parallel Q)[\vec{x'}/\vec{x}]$ guaranteeing that the variables $\vec{x'} \in \mathcal{F}^{|\vec{x}|}$ are fresh in the sense above.

Mapping local variables into fresh variables. Recall that in the process of Skolemization in first-order logic, existentially quantified variables are replaced by functions depending on the variables bound by universal quantifiers preceding the existential quantifier. Similarly, in the process $(\mathbf{local}\,\vec{x}; c)\,Q$ we shall syntactically substitute the variables $\vec{x}$ by a term of the form $f_{\vec{x}}(\cdot)$. This term denotes a function that takes as argument the variables bound in c and Q by **abs** operators and return a vector $\vec{x}'$ of fresh variables.

Let us illustrate this situation with an example. Assume the following process

$$P = (\mathbf{abs}\ x; c)\,(\mathbf{local}\,y)\,\mathbf{tell}(\mathtt{out}(x, y))$$

For each term t such that the current store entails $c[t/x]$, the process P must have a new fresh variable y' and then execute $\mathbf{tell}(\mathtt{out}(t, y'))$. Let $f_y : \mathcal{T} \to \mathcal{F}$ be a function from

terms in the constraint system to fresh variables. We get rid of the **local** process in P by syntactically replacing y by the *term* $f_y(x)$ (*rather than the application of* f_y *to* x), notation $\{f_y(x)/y\}$. We then obtain $P' = (\mathbf{abs}\ x; c)\,\mathbf{tell}(\mathtt{out}(x, f_y(x)))$. As expected, if the current store entails $c[t/x]$, the process $\mathbf{tell}(\mathtt{out}(t, f(t)))$ is executed. Decreeing f_y to be injective, we guarantee that $f_y(t) \neq f_y(t')$ for two different terms t and t'.

Remind that $\vec{x};\vec{y}$ denotes the concatenation of the vector $\vec{x}$ and $\vec{y}$ (Convention 3.1.1). The following definition formalizes this idea.

Definition 4.4.1. *Let P be a* `utcc` *process in the normal form of Definition 4.3.1 without replications and $\mathsf{FL}[\![\cdot]\!]_{\vec{x}} : Proc \times \mathcal{P}(\mathcal{V}) \rightharpoonup Proc$ be defined as:*

$$\mathsf{FL}[\![P]\!]_{\vec{x}} = \begin{cases} \mathbf{skip} & \text{if} \quad P = \mathbf{skip} \\ \mathbf{tell}(c) & \text{if} \quad P = \mathbf{tell}(c) \\ \mathsf{FL}[\![Q_1]\!]_{\vec{x}} \parallel \mathsf{FL}[\![Q_2]\!]_{\vec{x}} & \text{if} \quad P = Q_1 \parallel Q_2 \\ (\mathbf{abs}\ \vec{y}; c)\,\mathsf{FL}[\![Q]\!]_{\vec{x};\vec{y}} & \text{if} \quad P = (\mathbf{abs}\ \vec{y}; c)\,Q \\ (\mathbf{tell}(c) \parallel \mathsf{FL}[\![Q]\!]_{\vec{x}})\{f_{\vec{y}}(\vec{x})/\vec{y}\} & \text{if} \quad P = (\mathbf{local}\,\vec{y}; c)\,Q \\ \mathbf{next}\,\mathsf{FL}[\![Q]\!]_{\vec{x}} & \text{if} \quad P = \mathbf{next}\,Q \\ \mathbf{unless}\ c\ \mathbf{next}\,\mathsf{FL}[\![Q]\!]_{\vec{x}} & \text{if} \quad P = \mathbf{unless}\ c\ \mathbf{next}\,Q \end{cases}$$

where $f_{\vec{y}}(\vec{x}) : \mathcal{T}^{|\vec{x}|} \to \mathcal{F}^{|\vec{y}|}$. The function $f_{\vec{y}}(\cdot)$ is assumed to be injective. Furthermore, given two such functions $f_{\vec{y}}$ and $f_{\vec{z}}$, we assume $ran(f_{\vec{y}}) \cap ran(f_{\vec{z}}) = \emptyset$.

Let us point out some details about the previous definition.

1. In $\mathsf{FL}[\![P]\!]_{\vec{x}}$ we require P to be in normal form. Then, the variables $\vec{y}$ in a process of the form $(\mathbf{local}\,\vec{y}; c)\,Q$ or $(\mathbf{abs}\ \vec{y}; c)\,Q$ do not occur quantified elsewhere (see Definition 4.3.1). This prevents us from replacing two different local variables by the same function. This also explains why it is not necessary the side condition $\vec{x} \cap \vec{y} = \emptyset$ in the rule for the abstraction operator in $\mathsf{FL}[\![\cdot]\!]$.

2. The function $\mathsf{FL}[\![P]\!]_{\vec{x}}$ is not defined for $P =! Q$. The reason is that it would be necessary to expand P into the infinite parallel composition $Q \parallel \mathbf{next}\,Q \parallel \mathbf{next}^2 Q \parallel \ldots$ to obtain a set of different fresh variables in each time unit. Take for example $R = \mathbf{tell}(c(y)) \parallel \mathbf{next}\,\mathbf{tell}(c(y))$ and $P = (\mathbf{local}\,y)\,R$ where $c(y)$ denotes a constraint c such that $y \in fv(c)$. The intuitive behavior of $!\,P$ is then to create a different local variable y in each time unit and then execute R with this *new* local variable. Assume we were to define

   $$\mathsf{FL}[\![!\,P]\!]_{\vec{x}} =! \mathsf{FL}[\![P]\!]_{\vec{x}}$$

   Under this definition, we obtain $\mathsf{FL}[\![!\,P]\!]_{\vec{x}} =! R\{f_y/y\}$ where a unique fresh variable (f_y) is created. Therefore, in the second time unit we observe only $c(f_y)$ instead of $c(y_1) \wedge c(y_2)$ for two different fresh variables y_1, y_2 as expected.

3. Finally, as intended, the process $Q = \mathsf{FL}[\![P]\!]_{\vec{x}}$ does not contain occurrences of **local** operators.

The following lemma states the correspondence between the behavior of the processes P and $\mathsf{FL}[\![P]\!]$.

Lemma 4.4.1. *Let $\mathsf{FL}[\![\cdot]\!]$ be as in Definition 4.4.1 and P be a process in the normal form of Definition 4.3.1 without replications.*

1. $P = P_1 \xrightarrow{(a_1,b_1)} P_2 \xrightarrow{(a_2,b_2)} ...P_i \xrightarrow{(a_i,b_i)}$ *if and only if*

$$\mathsf{FL}[\![P]\!]_\emptyset = P'_1 \xrightarrow{(a_1,b'_1)} P'_2 \xrightarrow{(a_2,b'_2)} ...P'_i \xrightarrow{(a_i,b'_i)}$$

where $b_j = \exists_{\mathcal{F}}(b'_j)$ *for* $0 < j \leq i$.

2. $P = P_1 \xrightarrow{(a_1,e_1)}_s P_2 \xrightarrow{(a_2,e_2)}_s P_3 \xrightarrow{(a_3,e_3)}_s \ldots$ *if and only if*

$$\mathsf{FL}[\![P]\!]_\emptyset = P'_1 \xrightarrow{(a_1,e'_1)}_s P'_2 \xrightarrow{(a_2,e'_2)}_s P'_3 \xrightarrow{(a_3,e'_3)}_s \ldots$$

where $e_i = \exists_{\mathcal{F}}(e'_i)$ *for* $i > 0$.

Proof. Let $P' = (\mathbf{local}\,\vec{y}; c)\,Q$ be a subterm of the process P. By construction of $\mathsf{FL}[\![P]\!]_\emptyset$, we can show that in $\mathsf{FL}[\![P']\!]_{\vec{x}} = P'' = (\mathbf{tell}(c) \parallel \mathsf{FL}[\![Q]\!]_{\vec{x}})\{f_{\vec{y}}(\vec{x})/\vec{y}\}$, the variables in the range of $f_{\vec{y}}$ do not appear elsewhere. The lemma then follows from the semantic correspondence of the standard local operator of CCP and the one defined by using fresh variables in [Mendler 1995, Nielsen 2002b].

□

4.4.2 Local-Free Fragment

In this section we describe a simpler normal form of processes without occurrences of local and replicated constructs. Then, we show that the output of this kind of processes can be easily characterized in both the operational and the symbolic semantics. This shall ease the proof of the forthcoming results.

The Local-Free Fragment. We say that a process P is in *local-free* normal form if P does not have occurrences of local operators and for all process of the form $(\mathbf{abs}\ \vec{x}; c)\,Q$ in P, Q is either $\mathbf{tell}(c)$ or $\mathbf{next}\,Q'$. More precisely,

Definition 4.4.2 (Local-Free Normal Form)**.** *We say that a process P is in* Local-Free Normal Form *if P takes the form*

$$P \equiv \mathbf{tell}(c) \parallel \prod_{j\in J} (\mathbf{abs}\ \vec{y}_j; c_j)\,\mathbf{tell}(d_j) \parallel \prod_{j'\in J'} (\mathbf{abs}\ \vec{y}'_j; c'_j)\,\mathbf{next}\,P'_j \parallel \prod_{k\in K} \mathbf{next}\,P_k \parallel \prod_{l\in L} \mathbf{unless}\ c_l\ \mathbf{next}\,P_l$$

where each P'_j,P_k and P_l are in local-free normal form.

The following lemma states that for all process without replication, an input-output equivalent process in local-free normal form can be found. In this lemma, we shall use the standard notion of process context: a process context is a process expression with a single hole, represented by $\mathcal{E}[\cdot]$, such that placing a process in the hole yields a well-formed process.

Lemma 4.4.2. *Let P be an abstracted-unless free process without local nor replicated processes. Then, there exists P' in local-free normal form such that for all context $\mathcal{E}[\cdot]$, $\mathcal{E}[P] \sim^{io} \mathcal{E}[P']$ and $\mathcal{E}[P] \sim^{io}_s \mathcal{E}[P']$.*

Proof. Let P be an abstracted-unless free process in the normal form of Definition 4.3.1 without **local** nor replicated processes:

$$P \equiv \prod_{i\in I} \mathbf{tell}(c_i) \parallel \prod_{j\in J} (\mathbf{abs}\ \vec{y}_j; c_j)\,P_j \parallel \prod_{k\in K} \mathbf{next}\,P_k \parallel \prod_{l\in L} \mathbf{unless}\ c_l\ \mathbf{next}\,P_l$$

We shall show that we can obtain P' in the desired normal form. Let $\mathcal{E}[\cdot]$ be an arbitrary context.

It is easy to see that $\mathcal{E}[\prod_{i\in I} \mathbf{tell}(c_i)] \sim^{io} \mathcal{E}[\mathbf{tell}(c)]$ and $\mathcal{E}[\prod_{i\in I} \mathbf{tell}(c_i)] \sim_s^{io} \mathcal{E}[\mathbf{tell}(c)]$ where $c = \bigwedge_{i\in I} c_i$. Notice that in the case $I = \emptyset$, we have $c = \mathtt{true}$.

Now we have to show that a process of the form $P' = (\mathbf{abs}\ \vec{x}; c)\, Q$ can be decomposed such that Q is either $\mathbf{tell}(c)$ or $\mathbf{next}\, Q'$. If Q is of the form $\mathbf{tell}(c)$ or $\mathbf{next}\, Q'$ we are done. If this is not the case, we have to consider two cases: $Q = Q_1 \parallel Q_2$ and $Q = (\mathbf{abs}\ \vec{y}; d)\, Q'$. We proceed separately for the case of the operational and the symbolic semantics.

$\sim^{io}$ Let $Q = Q_1 \parallel Q_2$. We can verify that $\mathcal{E}[P'] \sim^{io} \mathcal{E}[(\mathbf{abs}\ \vec{x}; c)\, Q_1 \parallel (\mathbf{abs}\ \vec{x}; c)\, Q_2]$ by noticing that if the current store entails $c[\vec{t}/\vec{x}]$, then both $Q_1[\vec{t_1}/\vec{x}]$ and $Q_2[\vec{t_1}/\vec{x}]$ are eventually executed.

Now assume $Q = (\mathbf{abs}\ \vec{y}; d)\, Q'$. Note that $Q'[\vec{t_1}/\vec{x}, \vec{t_2}/\vec{y}]$ is executed only if the current store entails both $c[\vec{t_1}/\vec{x}]$ and $d[\vec{t_1}, \vec{t_2}/\vec{x}, \vec{y}]$. Then, we can show that $\mathcal{E}[P'] \sim^{io} \mathcal{E}[(\mathbf{abs}\ \vec{x}; \vec{y}; c \wedge d)\, Q']$ assuming, by alpha conversion, that $\vec{y} \notin fv(c)$.

$\sim_s^{io}$ For the case $Q = Q_1 \parallel Q_2$, we use the fact that $\forall_{\vec{x}}(c \Rightarrow (c_1 \wedge c_2)) \equiv \forall_{\vec{x}}(c \Rightarrow c_1) \wedge \forall_{\vec{x}}(c \Rightarrow c_2)$ to show $\mathcal{E}[P'] \sim_s^{io} \mathcal{E}[(\mathbf{abs}\ \vec{x}; c)\, Q_1 \parallel (\mathbf{abs}\ \vec{x}; c)\, Q_2]$.

Now assume $Q = (\mathbf{abs}\ \vec{y}; d)\, Q'$. By alpha conversion we assume $\vec{y} \notin fv(c)$. We can use the fact that $\forall_{\vec{x}}(c \Rightarrow (\forall_{\vec{y}}(d \Rightarrow e))) \equiv \forall_{\vec{x}}\forall_{\vec{y}}(c \wedge d \Rightarrow e)$ (if $\vec{y} \notin fv(c)$) to show that $\mathcal{E}[P'] \sim_s^{io} \mathcal{E}[(\mathbf{abs}\ \vec{x}; \vec{y}; c \wedge d)\, Q']$.

□

Notice that in Definition 4.4.2 and then in Lemma 4.4.2 we do not consider replicated processes. This is due to two reasons. Firstly, our mechanism to remove local operators cannot handle replicated processes (Lemma 4.4.1). Secondly, in the general case, it is not possible to find the local-free normal form for replicated processes. To see this, take for example the local-free processes $R = (\mathbf{abs}\ y; d)\,!\,Q$ and $P = (\mathbf{abs}\ x; c)\,!\,R$. One can decompose P as $(\mathbf{abs}\ x; c)\, R \parallel (\mathbf{abs}\ x; c)\, \mathbf{next}\,!\,R$ and find the normal form for $(\mathbf{abs}\ x; c)\, R$. Nevertheless, $(\mathbf{abs}\ x; c)\, \mathbf{next}\,!\,R$ is not in normal form (since R is not in normal form). Then, we have to unfold $\mathbf{next}\,!\,R$ into $\mathbf{next}\ (R \parallel \mathbf{next}\,!\,R)$ where $\mathbf{next}\,!\,R$ is not, once again, in normal form.

Outputs in Normal Form Now we can characterize the (symbolic) outputs of a process in Local-Free normal form.

Lemma 4.4.3 (Output in Normal Form)**.** *Let P be an abstracted-unless free process without replicated nor local processes in local-free normal form:*

$$P \equiv \mathbf{tell}(c) \parallel \prod_{j\in J} (\mathbf{abs}\ \vec{y_j}; c_j)\, \mathbf{tell}(d_j) \parallel \prod_{j'\in J'} (\mathbf{abs}\ \vec{y'_j}; c'_j)\, \mathbf{next}\, P'_j \parallel \prod_{k\in K} \mathbf{next}\, P_k \parallel \prod_{l\in L} \mathbf{unless}\ c_l\ \mathbf{next}\, P_l$$

If $P \xrightarrow{(a,b)} Q$, then for all $j \in J$ and $j' \in J'$ there exists $\mathcal{T}_j \subseteq_{fin} \mathcal{T}^{|\vec{y_j}|}$, $\mathcal{T}'_j \subseteq_{fin} \mathcal{T}^{|\vec{y'_j}|}$ and $L' \subseteq L$ such that

$$b \equiv c \wedge \bigwedge_{j\in J} \left(\bigwedge_{\vec{t}\in\mathcal{T}_j} d_j[\vec{t}/\vec{y_j}] \right)$$

$$Q \equiv \prod_{j'\in J} \prod_{t'\in\mathcal{T}'_j} \left(P'_j[\vec{t'}/\vec{y'_j}] \right) \parallel \prod_{k\in K} P_k \parallel \prod_{l\in L'} P_l$$

Proof. The proof is immediate from the rules in Table 3.1. The sets $\mathcal{T}_j$ and $\mathcal{T}'_j$ represents the set of terms making valid the guards c_j and c'_j in $(\mathbf{abs}\ \vec{y_j}; c_j)\,\mathbf{tell}(d_j)$ and $(\mathbf{abs}\ \vec{y'_j}; c'_j)\,\mathbf{next}\,P'_j$ respectively. Notice that we can assume $\mathcal{T}_j$ and $\mathcal{T}'_j$ to be finite sets since we have an observable transition $P \xrightarrow{(a,b)} Q$. Finally, L' corresponds to the subset of **unless** processes whose guard c_l cannot be entailed from b. □

Similarly, we characterize the symbolic output of processes in Local-Free Normal Form.

Lemma 4.4.4 (Symbolic Output in Normal Form)**.** *Let P be an abstracted-unless free process without replicated nor local processes in local-free normal form*

$$P \equiv \mathbf{tell}(c) \parallel \prod_{j\in J} (\mathbf{abs}\ \vec{y_j}; c_j)\,\mathbf{tell}(d_j) \parallel \prod_{j'\in J'} (\mathbf{abs}\ \vec{y'_j}; c'_j)\,\mathbf{next}\,P'_j \parallel \prod_{k\in K} \mathbf{next}\,P_k \parallel \prod_{l\in L} \mathbf{unless}\ c_l\ \mathbf{next}\,P_l$$

If $P \xrightarrow{(a,e)}_s Q$, then there exists $L' \subseteq L$ such that

$$\begin{aligned} e &\equiv c \wedge \bigwedge_{j\in J} \left(\forall_{\vec{y_j}}\,(c_j \Rightarrow d_j)\right) \\ Q &\equiv \mathbf{tell}(\ominus e) \parallel \prod_{j'\in J} (\mathbf{abs}\ \vec{y_j}'; \ominus c'_j)\,P'_j \parallel \prod_{k\in K} P_k \parallel \prod_{l\in L'} P_l \end{aligned}$$

Proof. The proof is immediate from the rules in Table 4.1. □

Recall that a process of the form $(\mathbf{abs}\ \vec{x}; c)\,\mathbf{next}\,P$ does not exhibit any internal symbolic reduction since $\mathbf{next}\,P$ does not evolve during the current time unit. Nevertheless, in the operational semantics, the process above evolves into a set of processes of the form $\mathbf{next}\,P[\vec{t}/\vec{x}]$. This explains the difference between the residual process $Q = \prod_{j'\in J} \prod_{t'\in\mathcal{T}'_j} \left(P'_j[\vec{t'}/\vec{y'_j}]\right)$ in Lemma 4.4.3 and $Q' = \prod_{j'\in J} (\mathbf{abs}\ \vec{y_j}'; \ominus c'_j)\,P'_j$ in Lemma 4.4.4. The following lemma relates Q and Q' by showing that the symbolic outputs of both processes entail the same basic constraints.

Lemma 4.4.5. *Let S, R be abstracted-unless free processes of the form*

$$\begin{aligned} S &= \mathbf{tell}(\ominus(e)) \parallel \prod_{j\in J} (\mathbf{abs}\ \vec{y_j}; \ominus c_j)\,Q_j \parallel P \\ R &= \prod_{j\in J} \prod_{t\in\mathcal{T}_j} \left(Q_j[\vec{t}/\vec{y_j}]\right) \parallel P \end{aligned}$$

such that for all $j \in J$, $\mathcal{T}_j = \{\vec{t} \mid e \models_T c_j[\vec{t}/\vec{x_j}]\} \subseteq_{fin} \mathcal{T}^{|\vec{x_j}|}$. Assume that P, Q_j are processes without occurrences of past formulae and let α be a sequence of constraints. If $S \xrightarrow{(\alpha,w)}_s$ and $R \xrightarrow{(\alpha,w')}_s$, then for all basic constraint d and $i > 0$, $w(i) \models_T d$ iff $w'(i) \models_T d$.

Proof. For the sake of clarity and without loss of generality, we shall assume J to be a singleton. The following arguments straightforwardly extend to the general case $|J| > 1$.

Let $\alpha = a_1.a_2.a_3...$, $\mathcal{T}' = \{\vec{t} \mid e \models_T c[\vec{t}/\vec{y}]\} \subseteq_{fin} \mathcal{T}^{|\vec{x_j}|}$ and

$$\begin{aligned} S &= \mathbf{tell}(\ominus(e)) \parallel (\mathbf{abs}\ \vec{y}; \ominus(c))\,Q \parallel P \\ R &= \prod_{\vec{t}\in\mathcal{T}'} Q[\vec{t}/\vec{y}] \parallel P \end{aligned}$$

By alpha conversion we can assume $\vec{y} \notin fv(\alpha) \cup fv(P)$. Assume the following derivation of S and R

$$S = S_1 \xrightarrow{(a_1, d_1 \wedge g_1)}_s S_2 \xrightarrow{(a_2, d_2 \wedge g_2)}_s S_3 \xrightarrow{(a_3, d_3 \wedge g_3)}_s \dots$$
$$R = R_1 \xrightarrow{(a_1, d'_1 \wedge g'_1)}_s R_2 \xrightarrow{(a_2, d'_2 \wedge g'_2)}_s R_3 \xrightarrow{(a_3, d'_3 \wedge g'_3)}_s \dots$$

where $d_1, d_2, d_3...$ and $d'_1, d'_2, d'_3...$ correspond to the constraints output by the processes $\mathbf{tell}(\ominus(e)) \parallel (\mathbf{abs}\ \vec{y}; \ominus(c))\, Q$ and $\prod_{\vec{t} \in \mathcal{T}} Q[\vec{t}/\vec{y}]$ respectively; and $g_1.g_2.g_3...$ and $g'_1.g'_2.g'_3...$ correspond to the output of P in S and R, respectively.

Recall that the symbolic future function F_s transfers the final store e to the next time unit as $\ominus(e)$. Given that $\vec{y} \notin fv(\alpha) \cup fv(P)$, by the rule $\mathrm{R}_{\mathrm{ABS-SYM}}$ and the definition of F_s, there exists $f_1, f_2, ...$ such that

$$Q = Q_1 \xrightarrow{(a_1, f_1)}_s Q_2 \xrightarrow{(a_2, f_2)}_s Q_3 \xrightarrow{(a_3, f_3)}_s \dots$$

and for $i > 0$, $d_i = \ominus^{i-1} \ominus (e) \wedge \forall_{\vec{y}}(\ominus^{i-1} \ominus (c) \Rightarrow f_i) = \ominus^i(e) \wedge \forall_{\vec{y}}(\ominus^i(c) \Rightarrow f_i)$.

Since S, R are abstracted-unless free, then Q is monotonic. By Lemma 4.3.8 we can show that there exists $Q'_1, Q'_2, Q'_3...$ such that

$$\prod_{\vec{t} \in \mathcal{T}} Q[\vec{t}/\vec{y}] = Q'_1 \xrightarrow{(a_1, d'_1)}_s Q'_2 \xrightarrow{(a_2, d'_2)}_s Q'_3 \xrightarrow{(a_3, d'_3)}_s \dots$$

where $d'_i = \bigwedge_{\vec{t} \in \mathcal{T}'} f_i[\vec{t}/\vec{y}]$.

Given that P and Q do not have occurrences of past formulae, we must have that for $i > 0$, all the past-formulae in f_i and g_i are guarded by at most $i - 1$ past operators. By hypothesis $\mathcal{T}' = \{\vec{t} \mid e \models_T c[\vec{t}/\vec{y}]\} \subseteq_{fin} \mathcal{T}^{|\vec{y}|}$ and then, we can show that for all basic constraint d,

$$\ominus^i(e) \wedge \forall_{\vec{y}}(\ominus^i(c) \Rightarrow f_i) \models_T d \ \text{ iff } \ \bigwedge_{\vec{t} \in \mathcal{T}'} f_i[\vec{t}/\vec{y}] \models_T d$$

Hence $d_i \models_T d$ iff $d'_i \models_T d$. By Lemma 4.3.4 we deduce $g_i = g'_i$. We then conclude $d_i \wedge g_i \models_T d$ iff $d'_i \wedge g'_i \models_T d$. □

Summing up the previous results, we have characterized the (symbolic) output of processes without local nor replicated processes. We have proven that the seemingly different residual processes resulting from both semantics exhibit the same symbolic outputs. The final step is then to show that the basic constraints entailed from both the symbolic and the operational outputs are the same. The following lemma proves that fact to conclude then with the correspondence theorem.

Lemma 4.4.6. *Let P be a abstracted-unless free process without local nor replicated processes and a be a basic constraint. If $P \xrightarrow{(a,b)} Q$ and $P \xrightarrow{(a,e)}_s R$ then for all basic constraint c, $b \models_T c$ iff $e \models_T c$. Furthermore, for all sequence of constraints α, if $Q \xrightarrow{(\alpha,w)}_s$ and $R \xrightarrow{(\alpha,w')}_s$, then for $i > 0$ $w(i) \models_T c$ iff $w'(i) \models_T c$.*

Proof. By Lemma 4.4.2, there exist S in the following normal form

$$S = \ \mathbf{tell}(c_i) \ \parallel \prod_{j \in J} (\mathbf{abs}\ \vec{y}_j; c_j)\, \mathbf{tell}(d_j) \parallel \prod_{j' \in J'} (\mathbf{abs}\ \vec{y}'_j; c'_j)\, \mathbf{next}\, P'_j$$
$$\parallel \prod_{k \in K} \mathbf{next}\, P_k \parallel \prod_{l \in L} \mathbf{unless}\ c_l\ \mathbf{next}\, P_l$$

such that $S \sim^{io} P$ and $S \sim_s^{io} P$. By hypothesis, $P \xrightarrow{(a,b)} Q$ and $P \xrightarrow{(a,e)}_s R$. Since $S \sim^{io} P$ and $S \sim_s^{io} P$, by Lemmas 4.4.3 and 4.4.4, we have

$$\begin{array}{lcl} b & \equiv & c_i \wedge \bigwedge_{j \in J} \left(\bigwedge_{\vec{t} \in \mathcal{T}_j} d_j[\vec{t}/\vec{y_j}] \right) \\ Q & \equiv & \prod_{j' \in J} \prod_{t' \in \mathcal{T}'_j} \left(P'_j[\vec{t'}/\vec{y'_j}] \right) \parallel \prod_{k \in K} P_k \parallel \prod_{l \in L'} P_l \\ e & \equiv & c_i \wedge \bigwedge_{j \in J} \left(\forall_{y_j} \, (c_j \Rightarrow d_j) \right) \\ R & \equiv & \mathbf{tell}(\ominus e) \parallel \prod_{j' \in J} (\mathbf{abs}\ \vec{y_j}'; \ominus c'_j) \, P'_j \parallel \prod_{k \in K} P_k \parallel \prod_{l \in L''} P_l \end{array}$$

where for all $j \in J$, the set $\mathcal{T}_j \subseteq_{fin} \mathcal{T}^{|\vec{y_j}|}$ corresponds to the set of terms $\{\vec{t} \mid b \models c_j[\vec{t}/\vec{y_j}]\}$. One can prove that $b \models_T c$ iff $e \models_T c$ for all basic constraint c. Hence $L' = L''$.

Since each P_k and P_l do not have occurrences of past formulae, by Lemma 4.4.5 we conclude that for all sequence of basic constraints α, if $Q \xrightarrow{(\alpha,w)}_s$ and $R \xrightarrow{(\alpha,w')}_s$, then $w(i) \models_T c$ iff $w'(i) \models_T c$ for $i > 0$. □

Notice that in the two previous lemmata we used the notation $\models_T$ instead of $\models_{T(\Delta)}$. Recall that we write $\models_T$ when $\Delta = \emptyset$. The following observation justifies this fact.

Observation 4.4.1 (Entailment and the Empty Theory)**.** *When a set of axioms Δ is assumed, it is possible that the formula representing the symbolic output of a process may entail more basic constraints than the constraint output by the operational semantics. This is due to the fact that once a theory Δ is considered, a particular interpretation of the predicates is assumed. Then, one may have that $F \models_\Delta c$ by appealing to the axioms in Δ and not only to the classical inference rules in logic.*

Let for example Δ be the axioms in Peano arithmetic and let

$$P = \mathbf{when}\ x \geq y\ \mathbf{do}\ \mathbf{tell}(d) \parallel \mathbf{when}\ x < y\ \mathbf{do}\ \mathbf{tell}(d)$$

Operationally, we know that $P \xrightarrow{(\mathtt{true},\mathtt{true})}$ and symbolically $P \xrightarrow{(\mathtt{true},e)}_s$ where

$$e = (x \geq y \Rightarrow d) \wedge (x < y \Rightarrow d)$$

Using Δ, we know that $\neg(x \geq y) = x < y$ and $\neg(x < y) = x \geq y$. Therefore, $e \models_\Delta d$. It is worth noticing that if $\Delta = \emptyset$, from $\neg(x \geq y)$ is not possible to entail $x < y$ and then $e \not\models d$ as above.

Notice also that processes are only allowed to add basic constraints (see Definition 3.1.2). Then, a process cannot add a constraint of the form $\neg c$.

This phenomenon was also studied in [de Boer 1997] where the authors enriched the logic of the constraint system to allow for a correspondence between the programming constructs in CCP and logical expressions as e above. □

Taking into account the previous observation, in the following results we assume only constraint systems where the theory Δ is empty.

4.4.3 Semantic Correspondence.

Now we are ready to state the main result of this section: The semantic correspondence between the operational and the symbolic semantics.

Theorem 4.4.1 (Semantic Correspondence)**.** *Let P be an abstracted-unless free process. Suppose that $P = P_1 \xrightarrow{(a_1,b_1)} P_2 \xrightarrow{(a_2,b_2)} \dots P_i \xrightarrow{(a_i,b_i)}$ and $P = P_1' \xrightarrow{(a_1,e_1)}_s P_2' \xrightarrow{(a_2,e_2)}_s \dots P_i' \xrightarrow{(a_i,e_i)}$. Then for every basic constraint c and $j \in \{1,\dots,i\}$, $d_j \models c$ iff $e_j \models_T c$.*

Proof. Since we are considering the execution of P until the i-th time unit, we can unfold the processes of the form $!Q$ into $Q \parallel \mathbf{next}\, Q \parallel \dots \parallel \mathbf{next}^{\,i-1} Q$. By Lemma 4.4.1 we can find P' such that $P' \sim^{io} P$ and $P' \sim_s^{io} P$ without local processes. The proof follows directly by repeated applications of Lemma 4.4.6. □

4.5 Summary and Related Work

In this chapter we introduced a symbolic semantics for the `utcc` calculus aiming at solving the infinite-branching problem of the operational semantics when considering non well-terminated processes. This semantics, for all process, is able to produce an output regardless the input from the environment. The key idea is to represent finitely with temporal formulae the infinitely many constraints output by the operational semantics. This way, the behavior of non well-terminated process can be observed. We proved that for the case of well-terminated processes, the output of both semantics entail the same basic constraints.

An application of this semantics is given in Chapter 8 where we model security protocols as `utcc` processes. These processes are non well-terminated since the model of the attacker may generate an unbound number of messages (constraints). This then shows the relevance of the symbolic characterization of `utcc` processes.

Furthermore, in Chapter 7, we shall define the symbolic input-output relation of a process and we shall show that this relation is a closure operator [Scott 1982], i.e., an idempotent, extensive and monotonic function when P is a monotonic process. We then give a denotational semantics capturing the set of fixed points of such an operator and we show that the behavior of P can be compositionally described.

The material of this chapter was originally published as [Olarte 2008c].

Related Work. In [Boreale 1996] a *symbolic semantics* is introduced to deal with the infinite-branching reduction relation in the π-calculus caused by infinitely many substitutions. Similarly, the works in [Boreale 2001b, Fiore 2001] propose a symbolic semantics for the *spi*-calculus to give a compact representation of the traces generated by the execution of a protocol. Roughly speaking, boolean constraints over names represent conditions the transition must hold to take place.

In the context of CCP based languages, in [Buscemi 2008] a symbolic characterization of the cc-pi calculus [Buscemi 2007] is given. Recall that cc-pi is a language combining the name passing mechanisms of the π-calculus and the CCP model. The constraint systems in cc-pi relies on named c-semirings, i.e. c-semirings [Bistarelli 1997] enriched with a notion of support to express the relevant names of a constraint.

The key idea of the symbolic transition system in [Buscemi 2008] is to have labels specifying the minimum conditions that must hold in order for a transition to take place. For instance, a process of the form $P = \mathbf{ask}\ c\ \mathbf{then}\ Q$ under a store d exhibits a symbolic transition of the form $P \xrightarrow{c'} Q$ where c' is the least restrictive constraint allowing P to evolve. The constraint c is obtained by means of the division operator ($\div$) of the c-semiring i.e., $c' = d \div c$. This symbolic semantics allows the authors to define an efficient characterization of open bisimulation [Sangiorgi 1996] for cc-pi.

Unlike the works mentioned above, the symbolic semantics we propose here not only deal with the infinite-branching problem but also with temporal issues and divergent internal computation in the operational semantics. To our knowledge, our proposal is the first semantics in concurrency theory using temporal constraints as finite representations of substitutions.

Chapter 5

Temporal Logic Characterization of utcc Processes

In addition to the usual behavioral techniques from process calculi, CCP-based calculi enjoy a declarative view of processes based upon logic [Saraswat 1993, Saraswat 1994, Mendler 1995, Fages 2001, Nielsen 2002a, de Boer 1997]. This makes CCP a language suitable for both the specification and the implementation of programs. In this chapter, we show that the utcc calculus is a declarative model for concurrency: utcc processes can be seen, at the same time, as computing agents and first-order linear-time temporal logic formulae (FLTL) [Manna 1991]. We do this by presenting a compositional encoding from utcc processes into FLTL formulae. Then, we prove that the (symbolic) outputs of a process P entail the same basic constraints that the FLTL formula A corresponding to P. That is, the operational point of view of processes and their logic characterization correspond to each other.

The logical characterization of utcc processes we propose here allows for using well-established techniques from FLTL for reachability analysis of utcc processes. For example, we can verify if a given security protocol modelled in utcc can reach a state where a secrecy property is violated. We later illustrate this scenario in Chapter 8.

Furthermore, in Chapter 6, we shall present a theoretical application of the logical view of utcc processes as FLTL formulae: We shall prove the undecidability of the validity problem for the monadic fragment of FLTL without equality nor function symbols.

5.1 utcc processes as FLTL formulae

In this section we give a compositional encoding from utcc processes into FLTL formulae. We shall use the past-free fragment of the Pnueli's FLTL [Manna 1991] in Definition 2.4.1. More precisely, we shall use the formulae generated by the following syntax:

$$F, G, \ldots := c \mid F \wedge G \mid \neg F \mid \exists x F \mid \circ F \mid \Box F.$$

Recall that c is a basic constraint and the modalities $\circ F$ and $\Box F$ state, respectively, that F holds *next* and *always*. See Definition 2.4.2 for the semantics of this logic.

The logic characterization of utcc is based upon a compositional mapping $\mathsf{TL}[\![\cdot]\!]$ from processes to FLTL formulae given below.

Definition 5.1.1. *Let* $\mathsf{TL}[\![\cdot]\!]$ *be a map from* ***utcc*** *processes to FLTL formulae given by:*

$$\mathsf{TL}[\![P]\!] = \begin{cases} \texttt{true} & \text{if} \quad P = \mathbf{skip} \\ c & \text{if} \quad P = \mathbf{tell}(c) \\ \forall \vec{y}(c \Rightarrow \mathsf{TL}[\![Q]\!]) & \text{if} \quad P = (\mathbf{abs}\ \vec{y}; c)\, Q \\ \mathsf{TL}[\![Q_1]\!] \wedge \mathsf{TL}[\![Q_2]\!] & \text{if} \quad P = Q_1 \parallel Q_2 \\ \exists \vec{x}(c \wedge \mathsf{TL}[\![Q]\!]) & \text{if} \quad P = (\mathbf{local}\ \vec{x}; c)\, Q \\ \circ \mathsf{TL}[\![Q]\!] & \text{if} \quad P = \mathbf{next}\, Q \\ c \vee \circ \mathsf{TL}[\![Q]\!] & \text{if} \quad P = \mathbf{unless}\ c\ \mathbf{next}\, Q \\ \Box \mathsf{TL}[\![Q]\!] & \text{if} \quad P = !\, Q \end{cases}$$

Let us give some intuitions about the previous encoding. Since **skip** cannot add any constraint, its corresponding formula is `true`. Similarly, $\mathbf{tell}(c)$ can only output c. Then, we map this process to the formula c.

The process $P = (\mathbf{abs}\ x; c)\, Q$ executes $Q[\vec{t}/\vec{x}]$ such that $c[\vec{t}/\vec{x}]$ can be entailed from the current store. Then, basic constraints that can be deduced from the output of P must correspond to formulae of the form $\mathsf{TL}[\![Q]\!][\vec{t}/\vec{x}]$. Therefore, we map P to the universally quantified formula $\forall_{\vec{x}}(c \Rightarrow \mathsf{TL}[\![Q]\!])$.

The parallel composition $P_1 \parallel P_2$ is mapped to the conjunction $A = \mathsf{TL}[\![P_1]\!] \wedge \mathsf{TL}[\![P_2]\!]$: A constraint c can be entailed from A if and only if the output produced by the interaction of P_1 and P_2 can entail c.

The local process $P = (\mathbf{local}\, \vec{x}; c)\, Q$ is dual to the abstraction. It corresponds to the existentially quantified formula $A = \exists_{\vec{x}}(c \wedge \mathsf{TL}[\![Q]\!])$. The intuition is that the output of the processes P and $P' = \mathbf{tell}(c) \parallel Q$ differ only in that P hides the information produced on the variables in $\vec{x}$.

For the case of the temporal constructs, let $A = \mathsf{TL}[\![Q]\!]$. We map the process $\mathbf{next}\, Q$ to the formula $\circ A$. If $P = \mathbf{unless}\ c\ \mathbf{next}\, Q$, either the guard c holds in the current time interval or the formula A must hold in the next time interval. Then P is mapped to $c \vee \circ A$. Finally, the replication $!\, Q$ is mapped to $\Box A$, meaning that A must hold in all time intervals.

5.2 FLTL Correspondence

This section is devoted to proving the relation between the symbolic outputs of P and the basic constraints entailed by its corresponding formula $A = \mathsf{TL}[\![P]\!]$. Recall that we say that P eventually outputs c, notation $P \Downarrow_s^c$, if P exhibits a sequence of observable transitions of the form $P = P_1 \xrightarrow{(\texttt{true}, e_1)}_s P_2 \xrightarrow{(\texttt{true}, e_2)}_s \ldots . P_i \xrightarrow{(\texttt{true}, e_i)}_s$ and $e_i \models_T c$ (see Definition 4.2.1). Recall also that the modality $\Diamond F$ is an abbreviation of $\neg\Box\neg F$, intuitively meaning that *eventually* the formula F holds. Roughly speaking, we shall prove that for any basic constraint c, $P \Downarrow_s^c$ if and only if $A \models_T \Diamond c$.

We shall also extend this result for the operational semantics when considering well-terminated processes.

5.2.1 Symbolic Reductions Correspond to FLTL Deductions

We start by proving that symbolic reductions correspond to logic deductions. More precisely,

Lemma 5.2.1. *Let* $\mathsf{TL}[\![\cdot]\!]$ *be as in Definition 5.1.1 and* P *be an abstracted-unless free process. If* $\langle P, e\rangle \longrightarrow_s \langle P', e'\rangle$ *then* $\mathsf{TL}[\![P]\!] \wedge e \models_T \mathsf{TL}[\![P']\!] \wedge e'$.

Proof. The proof proceeds by induction on (the depth of) the inference of $\langle P, e\rangle \longrightarrow_s \langle P', e'\rangle$.

- Using $\mathrm{R}_{\mathrm{TELL}}$: In this case, $P = \mathbf{tell}(c)$, $P' = \mathbf{skip}$ and $e' = e \wedge c$. We then trivially have $e \wedge c \models_T e \wedge c$.

- Using $\mathrm{R}_{\mathrm{PAR}}$: Then $P = Q \parallel R$ and $P' = Q' \parallel R$ where $\langle Q, e\rangle \longrightarrow_s \langle Q', e'\rangle$ by a shorter inference. We know by induction that $\mathsf{TL}[\![Q]\!] \wedge e \models_T \mathsf{TL}[\![Q']\!] \wedge e'$. By definition, $\mathsf{TL}[\![Q \parallel R]\!] = \mathsf{TL}[\![Q]\!] \wedge \mathsf{TL}[\![R]\!]$. We then conclude $\mathsf{TL}[\![Q]\!] \wedge \mathsf{TL}[\![R]\!] \wedge e \models_T \mathsf{TL}[\![Q']\!] \wedge \mathsf{TL}[\![R]\!] \wedge e'$.

- Using $\mathrm{R}_{\mathrm{ABS}}$: Then $P = (\mathbf{abs}\ \vec{x}; c)\, Q$, $P' = (\mathbf{abs}\ \vec{x}; c)\, Q'$ and $e' = e \wedge \forall_{\vec{x}}(c \Rightarrow d)$. By alpha conversion we can assume $\vec{x} \notin fv(e)$ and then we have the following derivation

  $$\langle Q, e\rangle \longrightarrow_s \langle Q', e \wedge d\rangle$$

  By inductive hypothesis we have $\mathsf{TL}[\![Q]\!] \wedge e \models_T \mathsf{TL}[\![Q']\!] \wedge e \wedge d$.

  One can prove that given F, F' s.t. $F \models_T F'$, it must be the case that $\forall_{\vec{x}}(c \Rightarrow F) \models_T \forall_{\vec{x}}(c \Rightarrow F')$. We then derive that

  $$\forall_{\vec{x}}(c \Rightarrow (\mathsf{TL}[\![Q]\!] \wedge e)) \models_T \forall_{\vec{x}}(c \Rightarrow (\mathsf{TL}[\![Q']\!] \wedge e \wedge d))$$

  Since $\vec{x} \notin fv(e)$, one can easily show that $e \models_T \forall_{\vec{x}}(c \Rightarrow e)$. Using the equation above, we can deduce the following

  $$\forall_{\vec{x}}(c \Rightarrow \mathsf{TL}[\![Q]\!]) \wedge e \models_T \forall_{\vec{x}}(c \Rightarrow (\mathsf{TL}[\![Q']\!] \wedge d)) \wedge e$$

  By definition, $\mathsf{TL}[\![(\mathbf{abs}\ \vec{x}; c)\, Q]\!] = \forall_{\vec{x}}(c \Rightarrow \mathsf{TL}[\![Q]\!])$. Then we conclude

  $$\mathsf{TL}[\![(\mathbf{abs}\ \vec{x}; c)\, Q]\!] \wedge e \models_T \mathsf{TL}[\![(\mathbf{abs}\ \vec{x}; c)\, Q']\!] \wedge e \wedge \forall_{\vec{x}}(c \Rightarrow d)$$

- Using $\mathrm{R}_{\mathrm{LOC}}$. We must have $P = (\mathbf{local}\, \vec{x}; c)\, Q$ and a derivation of the form

  $$\langle Q, c \wedge \exists_{\vec{x}} e\rangle \longrightarrow_s \langle Q', c' \wedge \exists_{\vec{x}} e\rangle$$

  Then, $e' = e \wedge \exists_{\vec{x}} c'$ and $P' = (\mathbf{local}\, \vec{x}; c')\, Q'$. By induction,

  $$\mathsf{TL}[\![Q]\!] \wedge c \wedge \exists_{\vec{x}} e \models_T \mathsf{TL}[\![Q']\!] \wedge c' \wedge \exists_{\vec{x}} e$$

  Therefore,

  $$\exists_{\vec{x}}(c \wedge \mathsf{TL}[\![Q]\!]) \wedge \exists_{\vec{x}} e \models_T \exists_{\vec{x}}(c' \wedge \mathsf{TL}[\![Q']\!]) \wedge \exists_{\vec{x}} e$$

  From the fact that $e \models_T \exists_{\vec{x}}(e)$, we derive the following

  $$\exists_{\vec{x}}(c \wedge \mathsf{TL}[\![Q]\!]) \wedge e \models_T \exists_{\vec{x}}(c' \wedge \mathsf{TL}[\![Q']\!]) \wedge e \wedge \exists_{\vec{x}}(c')$$

  By definition, $\mathsf{TL}[\![(\mathbf{local}\, \vec{x}; c)\, Q]\!] = \exists_{\vec{x}}(c \wedge \mathsf{TL}[\![Q]\!])$. Given that $e' = e \wedge \exists_{\vec{x}}(c')$, we conclude

  $$\mathsf{TL}[\![(\mathbf{local}\, \vec{x}; c)\, Q]\!] \wedge e \models_T \mathsf{TL}[\![(\mathbf{local}\, \vec{x}; c')\, Q']\!] \wedge e'$$

- Using $\mathrm{R}_{\mathrm{UNL}}$. In this case, $P = \mathbf{unless}\ c\ \mathbf{next}\, Q$, $e \models_T c$, $P' = \mathbf{skip}$ and $e' = e$. Then, trivially we have $\mathsf{TL}[\![P]\!] \wedge e \models_T e$.

- Using $\mathrm{R}_{\mathrm{REP}}$. Then $P = !\,Q$, $e = e'$ and $P' = Q \parallel \mathbf{next}\,!\,Q$. By definition of $\Box$, for all formula F, $\Box F \models_T F \wedge \circ\Box F$. Then we conclude

$$\Box\mathsf{TL}[\![Q]\!] \wedge e \models_T \mathsf{TL}[\![Q]\!] \wedge \circ\Box\mathsf{TL}[\![Q]\!] \wedge e$$

□

The previous lemma relates a single step of the symbolic internal reduction relation ($\longrightarrow_s$) with deductions in the FLTL. We extend this result to the symbolic observable relation ($\Longrightarrow_s$) in the next theorem.

Theorem 5.2.1. *Let $\mathsf{TL}[\![\cdot]\!]$ be as in Definition 5.1.1. Given a monotonic process P,*

(1) If $\langle P, e\rangle \longrightarrow_s^* \langle P', e'\rangle \not\longrightarrow_s$ then $\mathsf{TL}[\![P]\!] \wedge e \models_T \mathsf{TL}[\![P']\!] \wedge e'$

(2) If $P \xRightarrow{(e,e')}_s Q$ then $\mathsf{TL}[\![P]\!] \wedge e \models_T \circ\mathsf{TL}[\![Q]\!] \wedge e'$.

Proof. (1) is proved by repeated applications of Lemma 5.2.1.

For (2), let $e_1 = e$ and assume the following derivation of $P_1 = P$

$$\langle P_1, e_1\rangle \longrightarrow_s \langle P_2, e_2\rangle \longrightarrow_s \ldots \longrightarrow_s \langle P_n, e_n\rangle \not\longrightarrow_s$$

Then, $e' = e_n$ and $Q = F_s(P_n, e_n)$. From (1), we have $\mathsf{TL}[\![P_1]\!] \wedge e_1 \models_T \mathsf{TL}[\![P_n]\!] \wedge e_n$ and therefore $\mathsf{TL}[\![P]\!] \wedge e \models_T e'$. By Proposition 4.3.1, P_n takes the following normal form

$$P_n \equiv (\mathbf{local}\,\vec{x}; c)\left(\prod_{j\in J}(\mathbf{abs}\ \vec{x}_j; c_j)\,P_j \parallel \prod_{k\in K}\mathbf{next}\,P_k\right)$$

By case analysis of the function F_s one can easily prove that

$$\mathsf{TL}[\![P_n]\!] \wedge e_n \models_T \circ\mathsf{TL}[\![F_s(P_n, e_n)]\!]$$

□

Let us point out an important issue in the previous theorem.

Remark 5.2.1. *Recall that the process $P = \mathbf{unless}\ c\ \mathbf{next}\,Q$ is mapped to a formula of the form $F = c \vee \circ\mathsf{TL}[\![Q]\!]$. Notice that in general, from F, it is not possible to entail neither c nor $\circ\mathsf{TL}[\![Q]\!]$. Let us clarify this with a simple example. Assume $Q = \mathbf{tell}(d)$ and let e be a constraint such that $e \not\models_T c$. Then $P \xRightarrow{(e,e)}_s Q' \equiv \mathbf{tell}(d) \parallel \mathbf{tell}(\ominus(e))$. In this case, $F = \mathsf{TL}[\![P]\!] = c \vee \circ d$. We notice that from the fact that $e \not\models_T c$ we cannot conclude $e \wedge F \models_T \circ d$ and then it does not hold that $\mathsf{TL}[\![P]\!] \wedge e \models_T \circ(Q)$ – (2) in Theorem 5.2.1.*

This can be explained from the fact that $e \not\models_T c$ does not imply $e \models_T \neg c$ as we pointed out in Remark 3.2.1.

Consequently, we restricted the previous theorem to the monotonic fragment of `utcc`.

5.2.2 Deductions in FLTL correspond to Symbolic Reductions

Now we shall study the relation between the basic constraints entailed from $A = \mathsf{TL}[\![P]\!]$ and those entailed by the symbolic outputs of P. Notice that A may contain future modalities (i.e. $\Box, \circ$) while the output of P in a given time unit, say e', is a future-free formula. Then, to establish the relation between FLTL and symbolic reductions, we define a "projection" of A into a future free formula A' that replaces with `true` the subformulae guarded by more than a given number of next ("$\circ$"). More precisely,

Definition 5.2.1. *Let F be a past-free formula and Cut_F be defined as*

$$Cut_F(F,i) = \begin{cases} c & \text{if} \quad F = c \\ Cut_F(F_1,i) \otimes Cut_F(F_2,i) & \text{if} \quad F = F_1 \otimes F_2 \text{ and } \otimes \in \{\wedge, \vee, \Rightarrow\} \\ \texttt{true} & \text{if} \quad F = \circ F_1 \text{ and } i = 0 \\ \circ Cut_F(F_1, i-1) & \text{if} \quad F = \circ F_1 \text{ and } i > 0 \\ Cut_F(F_1,i) \wedge \circ Cut_F(\Box F_1, i-1) & \text{if} \quad F = \Box F_1 \\ \nabla Cut_F(F_1,i) & \text{if} \quad F = \nabla F_1 \text{ and } \nabla \in \{\exists_{\vec{x}}, \forall_{\vec{x}}\} \end{cases}$$

Intuitively, $Cut_F(F,n)$ replaces the formulae of the form $\Box F$ with a finite extension $F \wedge \circ F \wedge \circ^2 F .... \wedge \circ^n F$. Furthermore, it replaces all the subformulae guarded by more than n occurrences of the *next* modality ($\circ$) with `true`.

Let us give a simple example illustrating the function Cut_F.

Example 5.2.1 (Function Cut_F)**.** *Let $\texttt{out}_1(\cdot)$ and $\texttt{out}_2(\cdot)$ be as in Example 3.6.1 and let $P = !\,(\mathbf{abs}\ x; \texttt{out}_1(x))\,\mathbf{next}\,\mathbf{tell}(\texttt{out}_2(x))$ be a process that sends in the next time unit on channel $\texttt{out}_2$ every message received (in the current time unit) on channel $\texttt{out}_1$.*

We have $F = \mathsf{TL}[\![P]\!] = \Box(\forall_{\vec{x}}(\texttt{out}_1(x) \Rightarrow \circ\,\texttt{out}_2(x)))$. The formula $Cut_F(F,2)$ is obtained as follows:

$$\begin{aligned} Cut_F(F,2) \quad = \quad & \forall_{\vec{x}}(\texttt{out}_1(x) \Rightarrow \circ\,\texttt{out}_2(x)) \wedge \\ & \circ\ \forall_{\vec{x}}(\texttt{out}_1(x) \Rightarrow \circ\,\texttt{out}_2(x)) \\ & \circ\circ \forall_{\vec{x}}(\texttt{out}_1(x) \Rightarrow \texttt{true}) \wedge \circ\circ\circ\,\texttt{true} \end{aligned}$$

Simplifying the expression above we obtain:

$$Cut_F(F,2) = \ \forall_{\vec{x}}(\texttt{out}_1(x) \Rightarrow \circ\,\texttt{out}_2(x)) \wedge \circ\ \forall_{\vec{x}}(\texttt{out}_1(x) \Rightarrow \circ\,\texttt{out}_2(x))$$

□

The following theorem states that in a derivation of the form $P \xrightarrow{(e,e')}_s P'$, the output e' entails the formula $Cut_F(\mathsf{TL}[\![P]\!], 0)$. Furthermore it establishes the relation between the FLTL formulae corresponding to P and P'.

Theorem 5.2.2. *Let $\mathsf{TL}[\![\cdot]\!]$ be as in Definition 5.1.1 and P be an abstracted-unless free process. If $P \xrightarrow{(e,e')}_s P'$ then*

1. $e' \models_T Cut_F(\mathsf{TL}[\![P]\!], 0)$
2. $\circ\mathsf{TL}[\![P']\!] \models_T \mathsf{TL}[\![P]\!]$

Proof. Assume a derivation of the form $P \xrightarrow{(e,e')}_s P'$. We proceed by induction on the size of P. In each case we prove (1) and (2) above.

- $P = \mathbf{skip}$. Trivial
- $P = \mathbf{tell}(c)$. We must have the following derivation

  $$P \xrightarrow{(e,e\wedge c)}_s \mathbf{tell}(\ominus(e \wedge c))$$

  Let $F_1 = \mathsf{TL}[\![\mathbf{tell}(c)]\!] = c$ and $F_2 = \mathsf{TL}[\![\mathbf{tell}(\ominus(e \wedge c))]\!] = \ominus(e \wedge c)$. Then we have (1) $e \wedge c \models_T F_1$ and (2) $\circ(F_2) \models_T F_1$.

- $P = Q \parallel R$. For $Q = Q_1$ and $R = R_1$, we must have the following derivation:

$$\begin{array}{l}\langle Q_1, e\rangle \longrightarrow_s \langle Q_2, e \wedge d_2\rangle \longrightarrow_s^* \langle Q_n, e \wedge d_n\rangle \not\longrightarrow_s \\ \langle R_1, e\rangle \longrightarrow_s \langle R_2, e \wedge g_2\rangle \longrightarrow_s^* \langle R_m, e \wedge g_m\rangle \not\longrightarrow_s\end{array}$$

By Proposition 4.3.1, Q_n and R_m must have the following normal form:

$$\begin{array}{rl} Q_n \equiv & (\mathbf{local}\,\vec{x}; c_1)\left(\prod_{j\in J_1}(\mathbf{abs}\ \vec{x}_j; c_j)\,Q_j \parallel \prod_{k\in K_1}\mathbf{next}\,Q_k \parallel \prod_{l\in L_1}\mathbf{unless}\ c_l\ \mathbf{next}\,Q_l\right) \\ R_m \equiv & (\mathbf{local}\,\vec{x}; c_1')\left(\prod_{j\in J_2}(\mathbf{abs}\ \vec{x}_j; c_j)\,R_j \parallel \prod_{k\in K_2}\mathbf{next}\,R_k \parallel \prod_{l\in L_2}\mathbf{unless}\ c_l\ \mathbf{next}\,R_l\right)\end{array}$$

Since P is an abstracted-unless free process, by Lemma 4.3.3, it must be the case that

$$\begin{array}{ll}\langle Q_1 \parallel R_1, e\rangle \longrightarrow_s^* & \langle Q_n \parallel R_1, e \wedge d_n\rangle \longrightarrow_s^* \\ & \langle Q_n \parallel R_m, e \wedge d_n \wedge g_m\rangle \longrightarrow_s^* \\ & \langle Q_n' \parallel R_m', e \wedge d_n \wedge g_m\rangle \not\longrightarrow_s\end{array}$$

where Q_n' and R_m' are as Q_n and R_m respectively, but where some of the processes **unless** c **next** S evolve into **skip**. Hence, there exists $L_1' \subseteq L_1$ and $L_2' \subseteq L_2$ s.t.

$$\begin{array}{rl} Q_n' \equiv & (\mathbf{local}\,\vec{x}; c_1)\left(\prod_{j\in J_1}(\mathbf{abs}\ \vec{x}_j; c_j)\,Q_j \parallel \prod_{k\in K_1}\mathbf{next}\,Q_k \parallel \prod_{l\in L_1'}\mathbf{unless}\ c_l\ \mathbf{next}\,Q_l\right) \\ R_m' \equiv & (\mathbf{local}\,\vec{x}; c_1')\left(\prod_{j\in J_2}(\mathbf{abs}\ \vec{x}_j; c_j)\,R_j \parallel \prod_{k\in K_2}\mathbf{next}\,R_k \parallel \prod_{l\in L_2'}\mathbf{unless}\ c_l\ \mathbf{next}\,R_l\right)\end{array}$$

Notice that $Q \xRightarrow{(e,e\wedge d_n)}_s F_s(Q_n, e \wedge d_n)$ and $R \xRightarrow{(e,e\wedge g_n)}_s F_s(R_m, e \wedge g_n)$. By inductive hypothesis we then have $e \wedge d_n \models_T Cut_F(\mathsf{TL}[\![Q]\!], 0)$ and $e \wedge g_m \models_T Cut_F(\mathsf{TL}[\![R]\!], 0)$. Since $\mathsf{TL}[\![Q \parallel R]\!] = \mathsf{TL}[\![Q]\!] \wedge \mathsf{TL}[\![R]\!]$ we conclude

$$(1)\quad e \wedge d_n \wedge g_m \models_T Cut_F(\mathsf{TL}[\![Q \parallel R]\!], 0)$$

As for (2), by inductive hypothesis we also have the following

$$\begin{array}{l}\circ\mathsf{TL}[\![F_s(Q_n, e \wedge d_n)]\!] \models_T \mathsf{TL}[\![Q]\!] \\ \circ\mathsf{TL}[\![F_s(R_m, e \wedge g_m)]\!] \models_T \mathsf{TL}[\![R]\!]\end{array}$$

Then, by definition of the function F_s we derive

$$\circ\mathsf{TL}[\![F_s(Q_n \parallel R_m, e \wedge d_n \wedge g_m)]\!] \models_T \mathsf{TL}[\![Q \parallel R]\!]$$

We know by Lemma 4.3.3 that the derivation from $Q_n \parallel R_n$ into $Q_n' \parallel R_n'$ corresponds only to reductions of processes of the form **unless** c_l **next** Q_l into **skip**. Then it must be the case that $e \wedge d_n \wedge g_m \models_T c_l$ and therefore $e \wedge d_n \wedge g_m \models_T (c_l \vee G)$ for any G, in particular, $G = \circ\mathsf{TL}[\![Q_l]\!]$. Using the same reasoning for all $l \in L_1 - L_1'$ and $l' \in L_2 - L_2'$ we conclude

$$(2)\ \circ\mathsf{TL}[\![F_s(Q_n' \parallel R_m', e \wedge d_n \wedge g_m)]\!] \models_T \mathsf{TL}[\![Q \parallel R]\!]$$

- $P = (\mathbf{abs}\ \vec{x}; c)\, Q$. By alpha conversion assume that $\vec{x} \notin fv(e)$. We then must have the following evolution of $Q_1 = Q$

$$\langle Q_1, e\rangle \longrightarrow_s \langle Q_2, e \wedge d_2\rangle \longrightarrow_s^* \langle Q_n, e \wedge d_n\rangle \not\longrightarrow_s$$

We then have $Q \xrightarrow{(e, e\wedge d_n)}_s F_s(Q_n, e \wedge d_n)$ and by inductive hypothesis

$$(1)\ e \wedge d_n \models_T Cut_F(\mathsf{TL}[\![Q]\!], 0) \ \text{ and } \ (2)\ \ \circ\mathsf{TL}[\![F_s(Q_n, e \wedge d_n)]\!] \models_T \mathsf{TL}[\![Q]\!]$$

Since $\vec{x} \notin fv(e)$, by the Rule $\mathrm{R}_{\mathrm{ABS-SYM}}$ we have

$$\langle (\mathbf{abs}\ \vec{x}; c)\, Q, e\rangle \longrightarrow_s^* \langle (\mathbf{abs}\ \vec{x}; c)\, Q_n, e \wedge \forall_{\vec{x}}(c \Rightarrow d_n)\rangle \not\longrightarrow_s$$

For any formula F, F' s.t. $F \models_T F'$, we can prove that $\forall_{\vec{x}}(c \Rightarrow F) \models_T \forall_{\vec{x}}(c \Rightarrow F')$. Given that $e \wedge d_n \models_T Cut_F(\mathsf{TL}[\![Q]\!], 0)$ and $\vec{x} \notin fv(e)$ we deduce

$$e \wedge \forall_{\vec{x}}(c \Rightarrow d_n) \quad \models_T \quad \forall_{\vec{x}}(c \Rightarrow Cut_F(\mathsf{TL}[\![Q]\!], 0))$$

and by definition of Cut_F we conclude

$$(1)\ \ e \wedge \forall_{\vec{x}}(c \Rightarrow d_n) \models_T Cut_F(\forall_{\vec{x}}(c \Rightarrow \mathsf{TL}[\![Q]\!]), 0)$$

To prove (2), let F'_s be as in Definition 4.2.1 (symbolic future function). From $\vec{x} \notin fv(e)$ and $\circ\mathsf{TL}[\![F_s(Q_n, e \wedge d_n)]\!] \models_T \mathsf{TL}[\![Q]\!]$ we derive

$$\begin{array}{rcl} e \wedge d_n \wedge \circ\mathsf{TL}[\![F'_s(Q_n)]\!] & \models_T & \mathsf{TL}[\![Q]\!] \\ e \wedge (c \Rightarrow d_n) \wedge c \Rightarrow \circ\mathsf{TL}[\![F'_s(Q_n)]\!] & \models_T & c \Rightarrow \mathsf{TL}[\![Q]\!] \\ e \wedge \forall_{\vec{x}}(c \Rightarrow d_n) \wedge \forall_{\vec{x}}(c \Rightarrow \circ\mathsf{TL}[\![F'_s(Q_n)]\!]) & \models_T & \forall_{\vec{x}}(c \Rightarrow \mathsf{TL}[\![Q]\!]) \end{array}$$

From $\mathsf{TL}[\![(\mathbf{abs}\ \vec{x}; c)\, Q]\!] = \forall_{\vec{x}}(c \Rightarrow \mathsf{TL}[\![Q]\!])$ and the definition of F_s we conclude

$$(2)\ \ \circ\mathsf{TL}[\![F_s((\mathbf{abs}\ \vec{x}; c)\, Q_n, e \wedge \forall_{\vec{x}}(c \Rightarrow d_n))]\!] \models_T \mathsf{TL}[\![(\mathbf{abs}\ \vec{x}; c)\, Q]\!]$$

- $P = (\mathbf{local}\, \vec{x}; c)\, Q$. Let $Q_1 = Q$. We must have the following derivation:

$$\langle Q_1, \exists_{\vec{x}}(e) \wedge c\rangle \longrightarrow_s \langle Q_2, \exists_{\vec{x}}(e) \wedge c \wedge d_2\rangle \longrightarrow_s^* \langle Q_n, \exists_{\vec{x}}(e) \wedge c \wedge d_n\rangle$$

Therefore, $Q \xrightarrow{(e_1, e_2)}_s F_s(Q_n, e_2)$ where $e_1 = \exists_{\vec{x}}(e) \wedge c$ and $e_2 = \exists_{\vec{x}}(e) \wedge c \wedge d_n$. By inductive hypothesis we have

$$(1)\ \exists_{\vec{x}}(e) \wedge c \wedge d_n \models_T Cut_F(\mathsf{TL}[\![Q]\!], 0) \ \text{ and } \ (2)\ \ \circ\mathsf{TL}[\![F_s(Q_n, \exists_{\vec{x}}(e) \wedge c \wedge d_n)]\!] \models_T \mathsf{TL}[\![Q]\!]$$

By the Rule $\mathrm{R}_{\mathrm{LOC}}$ we have the following

$$\langle (\mathbf{local}\, \vec{x}; c)\, Q, e\rangle \longrightarrow_s^* \langle (\mathbf{local}\, \vec{x}; c \wedge d_n)\, Q_n, e \wedge \exists_{\vec{x}}(c \wedge d_n)\rangle \not\longrightarrow_s$$

From $\exists_{\vec{x}}(e) \wedge c \wedge d_n \models_T Cut_F(\mathsf{TL}[\![Q]\!], 0)$ we derive $\exists_{\vec{x}}(e) \wedge c \wedge d_n \models_T Cut_F(\mathsf{TL}[\![Q]\!], 0) \wedge c$ and then $\exists_{\vec{x}}(e) \wedge \exists_{\vec{x}}(c \wedge d_n) \models_T \exists_{\vec{x}}(Cut_F(\mathsf{TL}[\![Q]\!], 0) \wedge c)$. Since $e \models_T \exists_{\vec{x}}(e)$ we conclude

$$(1)\ \ e \wedge \exists_{\vec{x}}(c \wedge d_n) \models_T \exists_{\vec{x}}(c \wedge Cut_F(\mathsf{TL}[\![Q]\!], 0)) = Cut_F(\mathsf{TL}[\![P]\!], 0)$$

For (2), let F'_s be as in Definition 4.2.1 (symbolic future function). From $\circ\mathsf{TL}[\![F_s(Q_n, \exists_{\vec{x}}(e) \wedge c \wedge d_n)]\!] \models_T \mathsf{TL}[\![Q]\!]$ and $\mathsf{TL}[\![(\mathbf{local}\, \vec{x}; c)\, Q]\!] = \exists_{\vec{x}}(c \wedge \mathsf{TL}[\![Q]\!])$ we derive

$$\begin{array}{r} \exists_{\vec{x}}(e) \wedge c \wedge d_n \wedge \circ\mathsf{TL}[\![F'_s(Q_n)]\!] \models_T \mathsf{TL}[\![Q]\!] \wedge c \\ \exists_{\vec{x}}(e) \wedge \exists_{\vec{x}}(c \wedge d_n \wedge \circ\mathsf{TL}[\![F'_s(Q_n)]\!]) \models_T \exists_{\vec{x}}(\mathsf{TL}[\![Q]\!] \wedge c) \\ \circ\mathsf{TL}[\![F'_s((\mathbf{local}\, \vec{x}; c \wedge d_n)\, Q_n, \exists_{\vec{x}}(e))]\!]) \models_T \mathsf{TL}[\![(\mathbf{local}\, \vec{x}; c)\, Q]\!] \end{array}$$

Since $e \wedge \exists_{\vec{x}}(c \wedge d_n) \models_T \exists_{\vec{x}} e$ we conclude

$$(2)\ \ \circ\mathsf{TL}[\![F_s((\mathbf{local}\, \vec{x}; c \wedge d_n)\, Q_n, e \wedge \exists_{\vec{x}}(c \wedge d_n)]\!] \models_T \mathsf{TL}[\![(\mathbf{local}\, \vec{x}; c)\, Q]\!]$$

- $P = \mathbf{next}\, Q$. We must have a derivation of the form

$$P \xrightarrow{(e,e)}_s \mathbf{tell}(\ominus e) \parallel Q$$

Since $Cut_F(\circ\mathsf{TL}[\![Q]\!], 0) = \mathtt{true}$, we trivially have (1) $e \models_T Cut_F(\circ\mathsf{TL}[\![Q]\!], 0)$. We also trivially have (2) $\circ(\ominus(e) \wedge \mathsf{TL}[\![Q]\!]) \models_T \circ\mathsf{TL}[\![Q]\!]$.

- $P = \mathbf{unless}\ c\ \mathbf{next}\, Q$. We consider two cases
  - $e \models_T c$. Therefore $P \xrightarrow{(e,e)}_s \mathbf{tell}(\ominus e)$. Since $Cut_F(c \vee \circ\mathsf{TL}[\![Q]\!], 0) = (c \vee \mathtt{true}) = \mathtt{true}$, we trivially have (1) $e \models_T Cut_F(\mathsf{TL}[\![\mathbf{unless}\ c\ \mathbf{next}\, Q]\!], 0)$. Furthermore, $e \models_T c$ implies $e \models_T (c \vee G)$ for any G. Then we conclude (2) $\circ(\ominus e) \models_T (c \vee \circ\mathsf{TL}[\![Q]\!])$.
  - $e \not\models_T c$. Then we have $P \xrightarrow{(e,e)} \mathbf{tell}(\ominus e) \parallel Q$ and we proceed as in the case of $\mathbf{next}\, Q$.
- $P = !\, Q$. We must have the following derivation for Q

$$\langle Q, e\rangle \longrightarrow_s^* \langle Q', e'\rangle \not\longrightarrow_s$$

By inductive hypothesis we have

$$e' \models_T \mathsf{TL}[\![Q]\!] \quad \text{and} \quad \circ\mathsf{TL}[\![F_s(Q', e')]\!] \models_T \mathsf{TL}[\![Q]\!]$$

By the rule $\mathrm{R_{REP}}$ we must have

$$\langle !\, Q, e\rangle \longrightarrow_s \langle Q \parallel \mathbf{next}\, !\, Q, e\rangle\rangle \longrightarrow_s^* \langle Q' \parallel \mathbf{next}\, !\, Q, e'\rangle \not\longrightarrow_s$$

Since $Cut_F(\mathsf{TL}[\![\mathbf{next}\, !\, Q]\!], 0) = \mathtt{true}$ we conclude (1) $e' \models_T Cut_F(\mathsf{TL}[\![!\, Q]\!], 0)$. Finally, since $\circ\mathsf{TL}[\![F_s(Q', e')]\!] \models_T \mathsf{TL}[\![Q]\!]$ then

$$(2)\ \circ\mathsf{TL}[\![F_s(Q' \parallel \mathbf{next}\, !\, Q, e')]\!] \models_T \mathsf{TL}[\![Q]\!] \wedge \circ\mathsf{TL}[\![!\, Q]\!] = \mathsf{TL}[\![P]\!]$$

□

The Theorem 5.2.2 considers a single interaction of the process P with the environment. The following corollary extends this result by considering a sequence of interactions.

Corollary 5.2.1. *Let* $\mathsf{TL}[\![\cdot]\!]$ *be as in Definition 5.1.1 and* P *be an abstracted-unless free process. If* $P = P_1 \xrightarrow{(e_1,e_1')}_s P_2 \xrightarrow{(e_2,e_2')} P_3 \xrightarrow{(e_3,e_3')} \ldots$ *then for* $i > 0$

$$\circ^{i-1}(e_i') \models_T Cut_F(\mathsf{TL}[\![P]\!], i-1)$$

Proof. Let $P_1 = P$. If $i = 1$, from Theorem 5.2.2 we have

$$e_1' \models_T Cut_F(\mathsf{TL}[\![P_1]\!], 0)$$

For $i > 1$, by repeated applications of Theorem 5.2.2 and the definition of Cut_F we derive the following

$$\begin{array}{rcl} \circ^{i-1}\mathsf{TL}[\![P_i]\!] & \models_T & \mathsf{TL}[\![P_1]\!] \\ Cut_F(\circ^{i-1}\mathsf{TL}[\![P_i]\!], i-1) & \models_T & Cut_F(\mathsf{TL}[\![P_1]\!], i-1) \\ \circ^{i-1} Cut_F(\mathsf{TL}[\![P_i]\!], 0)] & \models_T & Cut_F(\mathsf{TL}[\![P_1]\!], i-1) \end{array}$$

By Theorem 5.2.2, $e_i' \models_T Cut_F(\mathsf{TL}[\![P_i]\!], 0)$ and we conclude

$$\circ^{i-1} e_i' \models_T Cut_F(\mathsf{TL}[\![P_1]\!], i-1)$$

□

Theorem 5.2.1 and Corollary 5.2.1 allow us to prove the desired correspondence between the symbolic outputs of P and the basic constraints entailed by the FLTL formula $\mathsf{TL}[\![P]\!]$.

Theorem 5.2.3. *Let $\mathsf{TL}[\![\cdot]\!]$ be as in Definition 5.1.1, $\Downarrow_s$ be as in Definition 4.2.1, P be an abstracted-unless free process and c be a basic constraint.*

1. *If there exists $k \geq 0$ such that $Cut_F(\mathsf{TL}[\![P]\!], k) \models_T \Diamond c$ then $P \Downarrow_s^c$.*
2. *If P is monotonic and $P \Downarrow_s^c$ then $\mathsf{TL}[\![P]\!] \models_T \Diamond c$.*

Proof. Let P be an abstracted-unless free process and assume the following derivation

$$P = P_1 \xrightarrow{(\mathtt{true}, e_1)}_s P_2 \xrightarrow{(\mathtt{true}, e_2)}_s ....P_i \xrightarrow{(\mathtt{true}, e_i)}_s \ldots$$

1. Assume that there exists $k \geq 0$ such that $F = Cut_F(\mathsf{TL}[\![P]\!], k)$ and $F \models_T \Diamond c$. Then it must be the case that $F \models_T \bigvee_{i \in 0..k} (\circ^i(c))$. Let $0 \leq i \leq k$ and assume that $F \models_T \circ^i(c)$. By Corollary 5.2.1 we know that $e_{k+1} \models_T \ominus^k F$ and then $e_{k+1} \models_T \ominus^{k-i}(c)$. Since the sequence $e_1.e_2...$ is a past-monotonic sequence, we conclude $e_{1+i} \models_T c$ and then $P \Downarrow_s^c$.
2. Assume that P is monotonic and $P \Downarrow_s^c$. Then there exists $i > 0$ s.t. $e_i \models_T c$. By repeated application of Theorem 5.2.1 we have $\mathsf{TL}[\![P]\!] \models_T \circ^{i-1}(e_i)$. From the fact that $e_i \models_T c$ we conclude $\mathsf{TL}[\![P]\!] \models_T \Diamond c$.

□

Relaying on the semantic correspondence in Theorem 4.4.1 we can straightforwardly extend the previous result to the case of the operational semantics.

Corollary 5.2.2 (FLTL Correspondence -SOS)**.** *Let $\mathsf{TL}[\![\cdot]\!]$ be as in Definition 5.1.1, $\Downarrow$ be as in Definition 3.7.1, P be a well-terminated and abstracted-unless free process and c be a basic constraint.*

1. *If there exists $k \geq 0$ such that $Cut_F(\mathsf{TL}[\![P]\!], k) \models_T \Diamond c$ then $P \Downarrow^c$.*
2. *If P is monotonic and $P \Downarrow^c$ then $\mathsf{TL}[\![P]\!] \models_T \Diamond c$.*

Proof. Directly from Corollary 5.2.3 and Theorem 4.4.1. □

5.3 Summary and Related Work

In this chapter we gave a logic characterization of `utcc` processes as formulae in the future-free fragment of the Pnueli's first-order linear-time temporal logic [Manna 1991]. We showed that the operational view of processes and the declarative one based upon FLTL correspond each other. Namely, we proved that the (symbolic) outputs of a process and the FLTL formula corresponding to P entail the same basic constraints. This logic characterization allows for reachability analysis of `utcc` processes using techniques from FLTL. For example, in Chapter 8 we shall verify that a process P modeling a flawed security protocol reaches a state where a secret is revealed. Furthermore, as a compelling application of the encoding of `utcc` processes into FLTL formulae, we shall prove in Chapter 6 the undecidability of the validity problem for the monadic fragment of FLTL without equality nor function symbols.

The material of this chapter was originally published as [Olarte 2008c].

Related Work. CCP-based languages have been shown to have a strong connection to logic that distinguishes this model from other formalisms for concurrency. In [Mendler 1995], this correspondence is deeply studied by showing that CCP processes can be viewed as logic formulae, constructs in the language as logical connectives and simulations (runs) as proofs.

In [de Boer 1997], a calculus for proving correctness of CCP programs is introduced. In this framework, the specification of the program is given in terms of a first-order formula. The authors pointed out that some problems arise when representing non-deterministic choices by disjunction and when considering the representation of this logical connective in the constraint system. For example, the constraint $x \geq 0$ does not really represent the disjunction $x = 0 \vee x > 0$ since $x \geq 0 \not\models x = 0$ nor $x \geq 0 \not\models x > 0$. Therefore, the logic of the constraint system is enriched to describe properties of constraints. Then, a property represented by a constraint is interpreted as the set of constraints that entails it. Consequently, logical operators are interpreted in terms of the corresponding set-theoretic operations. This way, a program P is said to satisfy a given property A if the set of all its outputs is a subset of the constraints defining A.

The results in [de Boer 1997] are extended and strengthened in [Nielsen 2002a], where a proof system for the `ntcc` calculus is proposed (a non-deterministic extension of `tcc`). Unlike [de Boer 1997], due to the temporal nature of `ntcc`, [Nielsen 2002a] considers computation along time units.

In [Saraswat 1994] the authors propose a proof system for `tcc` based on an intuitionistic logic enriched with a next operator. Judgements in the proof system have the form $A_1, ..., A_n \models A$ where $A_1, ..., A_n$ and A are agents (processes). Such judgements are valid if and only if the intersection of the denotations of the agents $A_1, ..., A_n$ is contained in the denotation of A; equivalently, any observation that can be made from the parallel system of agents $A_1, ..., A_n$ can also be made from A.

In the context of the π-calculus, in [Palamidessi 2006] it is shown that a logic interpretation as formulae in First-Order Logic can be given to persistent π processes. In this fragment of the π-calculus, all inputs and outputs are assumed to be replicated, much like in `utcc` every input (abstraction) and output (tell) is persistent during a time unit (we shall elaborate more on the relation between π's inputs-outputs and `utcc`'s abstractions-tells in Chapter 6). Using this logic characterization, a correspondence between barbed observability (output of a process) and logical consequence is proven similar to our logic correspondence in Theorem 5.2.3.

The necessity of performing rechability analysis of `utcc` processes motivated the development of the logic characterization here presented. Particularly, we were interested in verifying if a process P eventually exhibits certain output c. We then considered more appropriated to establish a correspondence between the operational semantics and the logic characterization rather than defining a proof system in the lines of [Saraswat 1994, Nielsen 2002a, de Boer 1997].

As future work, we plan to extend the proof system in [Nielsen 2002a] to consider the `utcc` abstraction operator and then, to cope with judgements of the form $P \vdash_T A$ where A is a past-free formula. The meaning of this judgment is that every possible output of P is a model for the formula A. Notice that in [Nielsen 2002a] the underlying logic is CLTL (a temporal logic where formulae are interpreted on sequences of constraints). Here, the semantics of FLTL formulae is given in terms of sequences of states as described in Section 2.4. Both semantics are related as it was shown in [Valencia 2005, Lemma 5.4].

Chapter 6

Expressiveness of utcc and Undecidability of FLTL

In the previous chapters we have studied the semantics of utcc and its declarative view of processes as first-order linear time temporal-logic (FLTL) formulae. This chapter is devoted to studying the computational expressiveness of utcc. As an application of this study, we state a noteworthy decidability result for FLTL. Namely, the undecidability of the validity problem for monadic FLTL without equality nor function symbols.

The computational *expressiveness* of tcc languages has been thoroughly studied in the literature [Saraswat 1994, Valencia 2005, Tini 1999]. This allowed for a better understanding of tcc and its relation with other formalisms. In particular, the expressiveness studies in [Saraswat 1994, Valencia 2005] show that tcc processes can be represented as *finite-state* Büchi automata [Buchi 1962] and thus cannot encode Turing-powerful formalisms. In contrast, here we show that well-terminated utcc processes (see Definition 3.8.1) can encode formalisms such as *Minsky machines* and the λ-*calculus*. Although both formalisms are Turing-equivalent, these encodings serve different purposes.

On the one hand, the encoding of Minsky machines uses a very simple constraint system: the monadic fragment of first-order logic (FOL) without equality nor function symbols. It is well known that the validity and the satisfiability problems for this fragment are decidable (see e.g. [Borger 2001]). The utcc theory and the encoding of Minsky machines will allow us to prove that the same fragment in FLTL is strongly incomplete, and then, undecidable its validity problem. On the other hand, we provide a compositional encoding of the call-by-name λ-calculus but using a more involved constraint system. Namely, we use a constraint system with binary and ternary uninterpreted predicates. This encoding is a significant test of expressiveness since it shows that utcc is able to mimic one of the most notable and simple computational models achieving Turing completeness.

It is worth noticing that there are several works in the literature addressing the decidability of fragments of FLTL and in particular the monadic one [Abadi 1990, Merz 1992, Szalas 1988, Hodkinson 2000, Valencia 2005]. Our decidability result is insightful in that it answers an issue raised in a previous work and justifies some restrictions on monadic FLTL in other decidability results. More specifically, in [Valencia 2005] it was suggested that one could dispense with the restriction to negation-free formula in the decidability result for the FLTL fragment there studied. Our undecidability result actually contradicts this conjecture since with negation that logic would correspond to the FLTL here studied. Furthermore, the work in [Merz 1992] proves the decidability of monadic FLTL. This seemingly contradictory statement arises from the fact that unlike our result, [Merz 1992] disallows quantification over flexible variables. Our results, therefore, show that restriction to be necessary for decidability.

In summary, this chapter shows the full computational expressiveness of utcc, states new results in the decidability of monadic FLTL and clarifies previous decidability results and conjectures in the literature.

$$\text{M}-\text{INC}\ \frac{(l_i, \texttt{INC}(c_n, l_j)) \quad v'_n = v_n + 1 \quad v'_{1-n} = v_{1-n}}{(l_i, v_0, v_1) \longrightarrow_M (l_j, v'_0, v'_1)}$$

$$\text{M}-\text{DEC}\ \frac{(l_i, \texttt{DECJ}(c_n, l_j, l_k)) \quad v_n \neq 0 \quad v'_n = v_n - 1 \quad v'_{1-n} = v_{1-n}}{(l_i, v_0, v_1) \longrightarrow_M (l_k, v'_0, v'_1)}$$

$$\text{M}-\text{DECJ}\ \frac{(l_i, \texttt{DECJ}(c_n, l_j, l_k)) \quad v_n = 0}{(l_i, v_0, v_1) \longrightarrow_M (l_j, v_0, v_1)}$$

Figure 6.1: Reduction relation in Minsky machines.

6.1 Minsky Machines

We start our expressiveness study by presenting an encoding of Minsky machines [Minsky 1967] into monotonic well-terminated utcc processes. First, we briefly recall some basic definition of this computational model.

A two-counter Minsky machine M is an imperative program consisting of a sequence of labeled instructions $(l_1, L_1); \ldots; (l_n, L_n)$ which modify the values of two non-negative counters c_0 and c_1.

The instructions, using counters c_n for $n \in \{0, 1\}$, are of three kinds:

- $(l_i : \texttt{HALT})$: Halts the machine.
- $(l_i : \texttt{INC}(c_n, l_j))$: Increments the counter c_n and jumps to the instruction l_j.
- $(l_i : \texttt{DECJ}(c_n, l_j, l_k))$: Tests if c_n is zero and then jumps to the instruction l_j. If c_n is not zero, it jumps to l_k.

A *configuration* in a Minsky machine is a tuple (l_i, v_0, v_1) where l_i is the label of the instruction to be executed and v_0 and v_1 the current value of the counters. Evolutions between such configurations are described by the reduction relation $\longrightarrow_M$ in Figure 6.1. We shall use $\longrightarrow_M^*$ to denote the reflexive and transitive closure of $\longrightarrow_M$.

In the sequel, without loss of generality, we assume that counters are initially set to zero and the machine *starts* at the instruction l_1.

We say that a Minsky machine M *halts* if the control reaches the location of a HALT instruction. Furthermore, it *computes* the value n if it halts with $c_0 = n$.

Definition 6.1.1 (Minsky Machine Computations)**.** *Let M be a Minsky machine with instructions $(l_1, L_1); \ldots; (l_m; \texttt{HALT}); \ldots (l_n, L_n)$. Let $\longrightarrow_M$ be as in Figure 6.1. We say that M* halts *if there exists a derivation $(l_1, 0, 0) \longrightarrow_M^* (l_m, v_0, v_1) \not\longrightarrow_M$. Furthermore, if $v_0 = n$, we say that M computes the value n.*

6.2 Encoding Minsky Machines into utcc

We shall use in our encoding recursive definitions of the form $p(\vec{x}) \stackrel{\texttt{def}}{=} P$. Recall that in utcc they can be encoded as abstractions using a uninterpreted predicate $call_p(\cdot)$ of arity $\vec{x}$ (see Section 3.3.1).

Let us first introduce the constraint system we shall use for our encoding.

Counters:

$$
\begin{array}{lcl}
\texttt{ZERO}_n & \stackrel{\text{def}}{=} & \textbf{when } inc_n \textbf{ do next } (\textbf{local } a)\,(\texttt{NOT-ZERO}_n(a) \parallel \\
 & & \qquad\qquad !\,\textbf{when } \texttt{out}(a) \textbf{ do } \texttt{ZERO}_n) \parallel \\
 & & \textbf{when } idle_n \textbf{ do next } \texttt{ZERO}_n \parallel \\
 & & \textbf{tell}(isz_n) \\
\texttt{NOT-ZERO}_n(x) & \stackrel{\text{def}}{=} & \textbf{when } inc_n \textbf{ do next } (\textbf{local } b)\,(\texttt{NOT-ZERO}_n(b) \parallel \\
 & & \qquad\qquad !\,\textbf{when } \texttt{out}(b) \textbf{ do } \texttt{NOT-ZERO}_n(x)) \parallel \\
 & & \textbf{when } dec_n \textbf{ do next tell}(\texttt{out}(x)) \parallel \\
 & & \textbf{when } idle_n \textbf{ do next } \texttt{NOT-ZERO}_n(x) \parallel \\
 & & \textbf{tell}(\textit{not-zero}_n)
\end{array}
$$

Instructions:

$[\![(l_i : L_i)]\!]_I \stackrel{\text{def}}{=} \textbf{when } \texttt{out}(l_i) \textbf{ do } ins(l_i, L_i)$ where

$$
\begin{array}{lcl}
ins(l_i, \texttt{HALT}) & = & \textbf{tell}(\texttt{halt}) \parallel \textbf{next tell}(\texttt{out}(l_i)) \parallel \textbf{tell}(idle_0 \wedge idle_1) \\
ins(l_i, \texttt{INC}(c_n, l_j)) & = & \textbf{tell}(inc_n) \parallel \textbf{next tell}(\texttt{out}(l_j)) \parallel \textbf{tell}(idle_{1-n}) \\
ins(l_i, \texttt{DECJ}(c_n, l_j, l_k)) & = & \textbf{when } isz_n \textbf{ do } (\textbf{next tell}(\texttt{out}(l_j)) \parallel \textbf{tell}(idle_n)) \parallel \\
 & & \textbf{when } \textit{not-zero}_n \textbf{ do } (\textbf{tell}(dec_n) \parallel \textbf{next tell}(\texttt{out}(l_k))) \parallel \\
 & & \textbf{tell}(idle_{1-n})
\end{array}
$$

Figure 6.2: Encoding of Registers and Instructions . $n \in \{0, 1\}$

Constraint System for the encoding. We shall assume a very simple constraint system for our encoding. Namely, we shall use the monadic fragment of first-order logic without functions, nor equality. We presuppose the (monadic) predicates $\texttt{out}(\cdot)$ and $call\_not\text{-}zero(\cdot)$ (to encode the recursive procedure NOT-ZERO –see Definition 3.3.2). Furthermore, we assume the 0-adic predicates isz_n, inc_n, dec_n, $\textit{not-zero}_n$, $idle_n$ for $n \in \{0, 1\}$, halt and $call\_zero$ (to encode the recursive procedure ZERO).

Counters. The counters c_0 and c_1 initially set to 0 are obtained by replacing the subindex n in the definition of $\texttt{ZERO}_n$ with 0 and 1 respectively in Figure 6.2. Intuitively, isz_n is used to test if the counter is zero and inc_n and dec_n to trigger the actions of increment and decrement the counters respectively. The constraint $\textit{not-zero}_n$ in the store indicates that the value of the counter is not zero.

Each time an increment instruction is executed, a new local variable is created, say a, and the process $\texttt{NOT-ZERO}(a)$ executed. Decrement operations output these local variables on the global channel $\texttt{out}(\cdot)$. The process NOT-ZERO, when receiving the corresponding local variable on channel out, moves to the state immediately before the last increment instruction took place. If the counter is not currently used, i.e., if $idle_n$ can be deduced, the counter remains in the same state.

Instructions. For the set of instruction $(l_1, L_1); \ldots; (l_n, L_n)$ we assume a set of variables $l_1, \ldots, l_n$. By adding the constraint $\texttt{out}(l_i)$, the code of the instruction l_i is spawned. In the case of $(l_i, \texttt{HALT})$, the constraint `halt` is added to the current store. Furthermore, by adding the constraint $idle_0 \wedge idle_1$, we specify that both counters are idle. The operation $(l_i : \texttt{INC}(c_n, l_j))$ adds the constraint inc_n and then activates the instruction l_j in the next time unit. It also adds the constraint $idle_{1-n}$ to assert that the other counter is idle. Finally, the encoding of the instruction $(l_i : \texttt{DECJ}(c_n, l_j, l_k))$ asks if the counter c_n is zero, i.e, if isz_n can be deduced from the current store. If it is the case, then it activates in the next time unit the instruction l_j. If the constraint $not\text{-}zero_n$ can be deduced (i.e. $c_n > 0$), then the encoding of this instruction adds the constraint dec_n and activates the instruction l_k in the next time unit.

The following definition makes use of the processes in Figure 6.2 to define the encoding of a Minsky machine in `utcc`.

Definition 6.2.1 (Encoding of Minsky Machines into `utcc`)**.** *Let M be a Minsky machine with instructions $(l_1 : L1), ..., (l_n : L_n)$. Let $[\![\cdot]\!]_I$ be as in Figure 6.2 and*

$$\mathit{DEFS} = \ulcorner \mathit{ZERO}_0 \urcorner \parallel \ulcorner \mathit{ZERO}_1 \urcorner \parallel \ulcorner \mathit{NOT\text{-}ZERO}_0(x) \urcorner \parallel \ulcorner \mathit{NOT\text{-}ZERO}_1(x) \urcorner$$

where $\ulcorner \cdot \urcorner$ is the encoding of recursion in Definition 3.3.2. The encoding $\mathsf{M}[\![\cdot]\!]$ is defined as:

$$\mathsf{M}[\![M]\!] \quad = \quad (\textbf{local}\, l_1, \ldots, l_n)\, (\ \textbf{tell}(\texttt{out}(l_1)) \parallel \prod_{i \in \{1,..,n\}} !\,[\![(l_i : L_i)]\!]_I \parallel \mathit{ZERO}_0 \parallel \mathit{ZERO}_1 \parallel \mathit{DEFS})$$

where ZERO_0, ZERO_1, $\mathit{NOT\text{-}ZERO}_0$ and $\mathit{NOT\text{-}ZERO}_1$ are obtained by replacing the sub-index n by 0 and 1 respectively in the definition of ZERO_n and $\mathit{NOT\text{-}ZERO}_n$ in Figure 6.2

Notice that in the definition of $\mathsf{M}[\![\cdot]\!]$, the first instruction (l_1) is activated by adding the constraint $\texttt{out}(l_1)$. Furthermore, both counters are initially set to zero.

6.2.1 Representation of Numbers in `utcc`

As hinted at above, increment operations create a local name, say a, and then execute the process $\texttt{NOT-ZERO}(a)$. For the decrement operations, these local names are sent back on channel `out`. Then, the encoding moves to the state immediately before the last increment operation took place. In the following definition, we give a characterization of the state of the counters that makes more precise this idea.

Definition 6.2.2 (Numbers in `utcc`)**.** *Let c_n be a counter and DEFS be as in Definition*

6.2.1. Let us define $[\![c_n = k]\!]_N = [\![c_n = k]\!]'_N \parallel$ *DEFS where*

$$
\begin{array}{lcl}
[\![c_n = 0]\!]'_N & \stackrel{\text{def}}{=} & \mathit{ZERO}_n \\
[\![c_n = 1]\!]'_N & \stackrel{\text{def}}{=} & (\mathbf{local}\, a_1)\,(!\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathit{ZERO}_n \parallel \\
 & & \qquad \mathit{NOT\text{-}ZERO}_n(a_1)) \\
[\![c_n = 2]\!]'_N & \stackrel{\text{def}}{=} & (\mathbf{local}\, a_1, a_2)\,(!\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathit{ZERO}_n \parallel \\
 & & \qquad\qquad !\,\mathbf{when}\ \mathtt{out}(a_2)\ \mathbf{do}\ \mathit{NOT\text{-}ZERO}_n(a_1) \\
 & & \qquad\qquad \mathit{NOT\text{-}ZERO}_n(a_2)) \\
\ldots & & \\
[\![c_n = k]\!]'_N & \stackrel{\text{def}}{=} & (\mathbf{local}\, a_1, a_2, ..., a_k)\,(\\
 & & \qquad\qquad !\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathit{ZERO}_n \parallel \\
 & & \qquad\qquad !\,\mathbf{when}\ \mathtt{out}(a_2)\ \mathbf{do}\ \mathit{NOT\text{-}ZERO}_n(a_1)) \\
 & & \qquad\qquad \ldots \\
 & & \qquad\qquad !\,\mathbf{when}\ \mathtt{out}(a_k)\ \mathbf{do}\ \mathit{NOT\text{-}ZERO}_n(a_{k-1}) \\
 & & \qquad\qquad \mathit{NOT\text{-}ZERO}_n(a_k))
\end{array}
$$

The above construction realizes our intuition of the behavior of the encoding in Definition 6.2.1. The "state" $[\![c_n = 0]\!]_N$ is represented by the the process $\mathtt{ZERO}_n$ which adds the constraint isz_n to the current store. If no increment instruction is executed (i.e. the counter is idle), the process $\mathtt{ZERO}_n$ is executed in the next time unit and then the counter remains in zero.

A number $k > 0$ is represented by k local variables $a_1, .., a_k$ and the respective ask processes *waiting* for the reception of the corresponding variable on channel out to move to the previous state. For the case of $k = 1$, a decrement operation causes that $\mathtt{NOT\text{-}ZERO}_n(a_1)$ outputs the constraint $\mathtt{out}(a_1)$ in the next time unit. Therefore, $!\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n$ will execute the process $\mathtt{ZERO}_n$. Similarly, for the case $k > 1$, a decrement operation causes that the process $\mathtt{NOT\text{-}ZERO}_n(a_k)$ adds the constraint $\mathtt{out}(a_k)$ in the next time unit. Then, the process $!\,\mathbf{when}\ \mathtt{out}(a_k)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_{k-1})$ spawns $\mathtt{NOT\text{-}ZERO}_n(a_{k-1})$ and we obtain the state $[\![c_n = k - 1]\!]_N$.

In the presence of an increment operation, the process $\mathtt{NOT\text{-}ZERO}_n(a_k)$ creates a new local variable, say a_{k+1}, with the corresponding **when** process waiting for the reception of that variable. Furthermore, the process $\mathtt{NOT\text{-}ZERO}_n(a_{k+1})$ is executed and we obtain the state $[\![c_n = k + 1]\!]_N$.

Notation 6.2.1. *In the sequel, for the sake of presentation, we shall omit the process* DEFS *when presenting a derivation of a process of the form* $[\![c_n = k]\!]_N$.

6.2.2 Encoding of Machine Configurations

Using our definition of numbers, we can give a suitable mapping from configurations of the machine into utcc processes. This shall help us to prove the operational correspondence of the encoding.

Definition 6.2.3 (Encoding of Configurations)**.** *Let M be a Minsky machine with instructions $(l_1; L_1), ..., (l_n; L_n)$. Let $[\![\cdot]\!]_N$ be as in Definition 6.2.2 and $[\![\cdot]\!]_I$ be as in Figure 6.2. The encoding $[\![\cdot]\!]_C$ of a configuration of M is defined as*

$$
\begin{array}{lcl}
[\![(l_i, v_0, v_1)]\!]_C & \stackrel{\text{def}}{=} & (\mathbf{local}\, l_1, ..., l_n)\,(\ [\![c_0 = v_0]\!]_N \parallel [\![c_1 = v_1]\!]_N \parallel \\
 & & \qquad \mathbf{tell}(\mathtt{out}(l_i)) \parallel \prod_{i \in \{1,..,n\}} !\,[\![(l_i : L_i)]\!]_I\ )
\end{array}
$$

6.3 Correctness of the Encoding

This section is devoted to proving the correctness of the encoding above: We shall show that reductions of the Minsky machine M and the observable transitions of the utcc process $\mathsf{M}[\![M]\!]$ correspond to each other.

Before that, notice that the process $P = \mathsf{M}[\![M]\!]$ is not meant to be executed under the influence of any environment. In other words, we shall observe the transitions of P when the input is `true`. Then, for the sake of presentation, we shall use the following notation that ignores the observable outputs and assume the inputs to be `true`.

Notation 6.3.1. *We shall write $P_1 \longrightarrow P_2 \longrightarrow P_3 \ldots$ to denote the sequence of internal transitions $\langle P_1, c_1\rangle \longrightarrow \langle P_2, c_2\rangle \longrightarrow \langle P_3, c_3\rangle \ldots$ when $c_1 \equiv$ `true` and the constraints $c_2, c_3, \ldots$ are unimportant. Similarly, we shall write $P \Longrightarrow P'$ when $P \overset{(\texttt{true},c)}{\Longrightarrow} P'$ and c is unimportant. For any equivalence relation between processes $\sim$, if $P \Longrightarrow P'$ and $P' \sim Q$ we shall write $P \Longrightarrow\!\sim Q$.*

Residual Processes and Observables. The reader may have noticed that the processes of the form $Q = \mathbf{when}\ \texttt{out}(a_k)\ \mathbf{do}\ P$, waiting for the entailment of the constraint $\texttt{out}(a_k)$, appear replicated in Figure 6.2. This is because we cannot know a priori when the constraint $\texttt{out}(a_k)$ will be added to the store. We can show that once the process P in Q is executed, it is not executed again. In other words, after the execution of P, the process Q in the next time unit will remain inactive for the rest of the execution of the encoding.

Let us illustrate this with an example. We can show that $P = [\![c_n = 2]\!]_N \parallel \mathbf{tell}(dec_n)$ exhibits the following reductions:

$$
\begin{aligned}
P \quad \longrightarrow^* \quad & (\mathbf{local}\, a_1, a_2)\,(!\,\mathbf{when}\ \texttt{out}(a_1)\ \mathbf{do}\ \texttt{ZERO}_n \parallel !\,\mathbf{when}\ \texttt{out}(a_2)\ \mathbf{do}\ \texttt{NOT-ZERO}_n(a_1)) \\
& \quad \mathbf{when}\ inc_n\ \mathbf{do\ next}\ (\mathbf{local}\, b)\,(\texttt{NOT-ZERO}_n(b) \parallel \\
& \qquad\qquad !\,\mathbf{when}\ \texttt{out}(b)\ \mathbf{do}\ \texttt{NOT-ZERO}_n(a_2)) \parallel \\
& \quad \mathbf{when}\ dec_n\ \mathbf{do\ next\ tell}(\texttt{out}(a_2)) \parallel \\
& \quad \mathbf{when}\ idle_n\ \mathbf{do\ next}\ \texttt{NOT-ZERO}_n(a_2) \parallel \mathbf{tell}(not\text{-}zero_n) \parallel \mathbf{tell}(dec_n) \\
\longrightarrow^* \quad & (\mathbf{local}\, a_1, a_2)\,(!\,\mathbf{when}\ \texttt{out}(a_1)\ \mathbf{do}\ \texttt{ZERO}_n \parallel !\,\mathbf{when}\ \texttt{out}(a_2)\ \mathbf{do}\ \texttt{NOT-ZERO}_n(a_1)) \\
& \quad \mathbf{when}\ inc_n\ \mathbf{do\ next}\ (\mathbf{local}\, b)\,(\texttt{NOT-ZERO}_n(b) \parallel \\
& \qquad\qquad !\,\mathbf{when}\ \texttt{out}(b)\ \mathbf{do}\ \texttt{NOT-ZERO}_n(a_2)) \parallel \\
& \quad \mathbf{next\ tell}(\texttt{out}(a_2)) \parallel \\
& \quad \mathbf{when}\ idle_n\ \mathbf{do\ next}\ \texttt{NOT-ZERO}_n(a_2) \not\longrightarrow
\end{aligned}
$$

Then we have $P \Longrightarrow Q$ where

$$
\begin{aligned}
Q \quad \equiv \quad & (\mathbf{local}\, a_1, a_2)\,(!\,\mathbf{when}\ \texttt{out}(a_1)\ \mathbf{do}\ \texttt{ZERO}_n \parallel !\,\mathbf{when}\ \texttt{out}(a_2)\ \mathbf{do}\ \texttt{NOT-ZERO}_n(a_1)) \\
& \quad \parallel \mathbf{tell}(\texttt{out}(a_2))
\end{aligned}
$$

and

$$
\begin{aligned}
Q \quad \longrightarrow^* \quad & (\mathbf{local}\, a_1, a_2)\,(!\,\mathbf{when}\ \texttt{out}(a_1)\ \mathbf{do}\ \texttt{ZERO}_n \parallel !\,\mathbf{when}\ \texttt{out}(a_2)\ \mathbf{do}\ \texttt{NOT-ZERO}_n(a_1) \\
& \quad \parallel \texttt{NOT-ZERO}_n(a_1)) \equiv Q'
\end{aligned}
$$

By a simple inspection of Q', it is easy to see that a_2 only occurs in the process $!\,\mathbf{when}\ \texttt{out}(a_2)\ \mathbf{do}\ \texttt{NOT-ZERO}_n(a_1))$ (as a guard). Therefore, neither the process $\texttt{NOT-ZERO}_n(a_1)$ nor $!\,\mathbf{when}\ \texttt{out}(a_1)\ \mathbf{do}\ \texttt{ZERO}_n$ can add to the store a constraint entailing $\texttt{out}(a_2)$.

One can thus show that after eliminating the "residual" process

$$!\,\mathbf{when}\ \texttt{out}(a_2)\ \mathbf{do}\ \texttt{NOT-ZERO}_n(a_1))$$

the behavior of Q' remains the same, more precisely, one obtains an output equivalent process to Q'.

Proposition 6.3.1. *Let $P \equiv [\![c_n = k]\!]_N$ for some k and $\sim^o$ be as in Definition 3.7.1. Assume $a \notin fv(P)$. Then for all process Q the following holds*

$$P \sim^o (\mathbf{local}\, a)\,(P \parallel !\,\mathbf{when}\ \mathtt{out}(a)\ \mathbf{do}\ Q)$$

Proof. Since $a \notin fv(P)$ one can show that $P \equiv [\![c_n = k]\!]_N$ cannot add a constraint c entailing $\mathtt{out}(a)$. Then, the process **when** $\mathtt{out}(a)$ **do** Q does not exhibit any internal transition. □

6.3.1 Derivations in the Minsky Machine and utcc Observables

In this section we prove the correspondence between derivations in the Minsky machine M and the derivations of the process $\mathsf{M}[\![M]\!]$. Before establishing this correspondence, we require some additional results.

The following proposition states that the encoding of the set of instructions adds in each time unit one and only one of the following constraints: inc_n, or dec_n or $idle_n$ for $n \in \{0, 1\}$

Proposition 6.3.2 (Counter Operations)**.** *Let M be a Minsky machine with instructions $(l_1 : L1), ..., (l_n : L_n)$. Let $[\![\cdot]\!]_I$ be as in Figure 6.2 and R be defined as:*

$$R \quad = \quad (\mathbf{local}\, l_1, \ldots, l_n)\,(\ \mathbf{tell}(\mathtt{out}(l_i)) \parallel \prod_{i\in\{1,..,n\}} !\,[\![(l_i : L_i)]\!]_I)$$

If $R \xrightarrow{(\mathtt{true},c)}$, then for $n \in \{0, 1\}$ these conditions hold

1. *$c \models inc_n$ implies $c \models idle_{1-n}$.*
2. *$c \models dec_n$ implies $c \models idle_{1-n}$.*
3. *$c \models idle_n$ implies $c \not\models inc_n$ and $c \not\models dec_n$.*

Proof. One can easily show that $\mathbf{tell}(\mathtt{out}(l_i))$ causes the execution of $ins(l_i, L_i)$ in R. By a simple analysis of the process $ins(l_i, L_i)$ one can verify the conditions above. □

The following proposition introduces an obvious fact on the encoding of numbers in utcc. Namely, if there are no increment or decrement instructions on a counter (i.e., it is idle) its value remains the same.

Proposition 6.3.3. *Let $ins(\cdot)$ be as in Figure 6.2, (l_i, L_i) be an instruction and $C_n = [\![c_n = k]\!]_N$ for some k. If L_i is a halt instruction or an instruction on counter n , there exists k' and l_i' such that*

$$\begin{aligned}&(\mathbf{local}\, l_1, \ldots, l_n)\,(\ [\![c_{1-n} = k'']\!]_N \parallel [\![c_n = k]\!]_N \parallel \mathbf{tell}(\mathtt{out}(l_i)) \parallel \prod_{i\in\{1,..,n\}} !\,[\![(l_i : L_i)]\!]_I)\\ &\Longrightarrow\\ &(\mathbf{local}\, l_1, \ldots, l_n)\,(\ [\![c_{1-n} = k'']\!]_N \parallel [\![c_n = k']\!]_N \parallel \mathbf{tell}(\mathtt{out}(l_i')) \parallel \prod_{i\in\{1,..,n\}} !\,[\![(l_i : L_i)]\!]_I)\end{aligned}$$

Proof. Assume that $R = ins(l_i, L_i)$. If L_i is a halt instruction or an instruction on counter c_n, then by Proposition 6.3.2 we know that R adds the constraint $idle_{1-n}$ and cannot add neither inc_{1-n} nor dec_{1-n}. Let $P = [\![c_{n-1} = k'']\!]$. Assume that $k'' = 0$. By a simple analysis on P we notice that only the guard of the process **when** $idle_{n-1}$ **do next** $\mathtt{ZERO}_{n-1}$ can be entailed and then the conclusion follows.

Assume now $k'' > n$. Then $P = [\![c_{1-n} = k'']\!]_N$ takes the form

$$\begin{array}{l}(\mathbf{local}\, a_1, a_2, ..., a''_k)\,(\\ \qquad !\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_{1-n} \parallel\\ \qquad !\,\mathbf{when}\ \mathtt{out}(a_2)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_{1-n}(a_1))\\ \quad \ldots\\ \qquad !\,\mathbf{when}\ \mathtt{out}(a''_k)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_{1-n}(a''_{k-1}))\\ \qquad \mathtt{NOT\text{-}ZERO}_{1-n}(a''_k)\end{array}$$

By a simple analysis of the process $\mathtt{NOT\text{-}ZERO}_{1-n}$ we can show that only the guard of the process $\mathbf{when}\ idle_{1-n}\ \mathbf{do\ next}\ \mathtt{NOT\text{-}ZERO}_{1-n}(a''_k)$ can be deduced and then the conclusion follows. □

Now we are ready to prove that derivations in the Minsky machine ($\longrightarrow_M$) correspond to observable derivations ($\Longrightarrow$) in utcc.

Lemma 6.3.1 (Completeness)**.** *Let M be a Minsky machine with instructions $(l_1; L_1), ..., (l_n; L_n)$, $[\![\cdot]\!]_C$ be as in Definition 6.2.3 and (l_i, v_0, v_1) be a configuration of M.*

$$\textit{If } (l_i, v_0, v_1) \longrightarrow_M (l'_i, v'_0, v'_1) \textit{ then } [\![(l_i, v_0, v_1)]\!]_C \Longrightarrow \sim^o [\![(l'_i, v'_0, v'_1)]\!]_C$$

Furthermore, if $(l_i, v_0, v_1) \not\longrightarrow_M$, (i.e. l_i is a HALT *instruction),*

$$[\![(l_i, v_0, v_1)]\!]_C \overset{(\mathtt{true}, c)}{\Longrightarrow} [\![(l_i, v_0, v_1)]\!]_C \textit{ and } c \models \mathtt{halt}$$

Proof. Assume $(l_i, v_0, v_1) \longrightarrow_M (l'_i, v'_0, v'_1)$ and let

$$P = [\![(l_i, v_0, v_1)]\!]_C \quad = \quad (\mathbf{local}\, l_1, ..., l_n)\,(\, C_0 \parallel C_1 \parallel \mathbf{tell}(\mathtt{out}(l_i)) \parallel \prod_{i \in \{1,..,n\}} !\,[\![(l_i : L_i)]\!]_I\,)$$

where $C_n = [\![c_n = v_n]\!]_N$ for $n \in \{0, 1\}$.

We shall prove that $P \Longrightarrow Q$ and $Q \sim^o [\![(l'_i, v'_0, v'_1)]\!]_C$. The proof proceeds by case analysis of the instruction L_i.

1. $(li : \mathtt{INC}(c_n, l_j))$: We trivially have that $\mathbf{tell}(\mathtt{out}(l_i))$ in parallel with the encoding of the set of instructions reduces to the process $\mathbf{tell}(inc_n) \parallel \mathbf{next}\,\mathbf{tell}(\mathtt{out}(l_j))$. By using Proposition 6.3.3 we can show that the process C_{1-n} remains the same in Q. Now we have to prove that C_n evolves into $C'_n = [\![c_n = v'_n]\!]_N$ with $v'_n = v_n + 1$.

   Let us consider first the case $P' = [\![c_n = 0]\!]_N \parallel \mathbf{tell}(inc_n)$. We can show that there is a derivation of the form

$$\begin{array}{rl} P' & \longrightarrow^* \ \mathbf{next}\,(\mathbf{local}\, a_1)\,(!\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n \parallel \mathtt{NOT\text{-}ZERO}_n(a_1)) \parallel\\ & \qquad \mathbf{when}\ idle_n\ \mathbf{do\ next}\ \mathtt{ZERO}_n \parallel \mathbf{tell}(isz_n)\\ & \longrightarrow^* \ \mathbf{next}\,(\mathbf{local}\, a_1)\,(!\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n \parallel \mathtt{NOT\text{-}ZERO}_n(a_1)) \parallel\\ & \qquad \mathbf{when}\ idle_n\ \mathbf{do\ next}\ \mathtt{ZERO}_n \not\longrightarrow \end{array}$$

   Therefore, we have the following observable transition:

$$P' \Longrightarrow (\mathbf{local}\, a_1)\,(!\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n \parallel \mathtt{NOT\text{-}ZERO}_n(a_1)) \equiv [\![c_n = 1]\!]_N$$

   Now consider the case $P' = [\![c_n = k]\!]_N \parallel \mathbf{tell}(inc_n)$ for $k > 0$. We must have the

following derivation

$$
\begin{array}{lll}
P' & \longrightarrow^* & (\mathbf{local}\, a_1, a_2, ..., a_k)\,(\\
& & \quad !\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n \parallel \\
& & \quad \ldots \\
& & \quad !\,\mathbf{when}\ \mathtt{out}(a_k)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_{k-1}) \parallel \\
& & \quad \mathbf{when}\ inc_n\ \mathbf{do\ next}\,(\mathbf{local}\, a_{k+1})\,(\mathtt{NOT\text{-}ZERO}_n(a_{k+1}) \parallel \\
& & \qquad\qquad !\,\mathbf{when}\ \mathtt{out}(a_{k+1})\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_k)) \parallel \\
& & \quad \mathbf{when}\ dec_n\ \mathbf{do\ next\ tell}(\mathtt{out}(a_k)) \parallel \\
& & \quad \mathbf{when}\ idle_n\ \mathbf{do\ next}\ \mathtt{NOT\text{-}ZERO}_n(a_k) \parallel \\
& & \quad \mathbf{tell}(\textit{not-zero}_n) \parallel \mathbf{tell}(inc_n)) \\
\\
& \longrightarrow^* & (\mathbf{local}\, a_1, a_2, ..., a_k)\,(\\
& & \quad !\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n \parallel \\
& & \quad \ldots \\
& & \quad !\,\mathbf{when}\ \mathtt{out}(a_k)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_{k-1}) \parallel \\
& & \quad \mathbf{next}\,(\mathbf{local}\, a_{k+1})\,(\mathtt{NOT\text{-}ZERO}_n(a_{k+1}) \parallel \\
& & \qquad\qquad !\,\mathbf{when}\ \mathtt{out}(a_{k+1})\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_k)) \parallel \\
& & \quad \mathbf{when}\ dec_n\ \mathbf{do\ next\ tell}(\mathtt{out}(a_k)) \parallel \\
& & \quad \mathbf{when}\ idle_n\ \mathbf{do\ next}\ \mathtt{NOT\text{-}ZERO}_n(a_k)) \not\longrightarrow
\end{array}
$$

We then have $P' \Longrightarrow\!\!\sim^o Q'$ where

$$
\begin{array}{lll}
Q' & \equiv & (\mathbf{local}\, a_1, a_2, ..., a_k, a_{k+1}) \\
& & \quad !\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n \parallel \\
& & \quad !\,\mathbf{when}\ \mathtt{out}(a_2)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_1) \parallel \\
& & \quad \ldots \\
& & \quad !\,\mathbf{when}\ \mathtt{out}(a_{k+1})\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_k) \parallel \\
& & \quad \mathtt{NOT\text{-}ZERO}_n(a_{k+1})) \\
& \equiv & [\![c_n = k+1]\!]_N
\end{array}
$$

We then conclude that $Q \sim^o [\![(l'_i, v'_0, v'_1)]\!]_C$.

2. $(li : \mathtt{DECJ}(c_n, l_j, l_k))$. Similar to the increment case, by using Proposition 6.3.3 we can show that the process C_{1-n} remains the same in Q.

   We then have to prove that $C_n = [\![c_n = v_n]\!]_N$ evolves into $C'_n = [\![c_n = v_n]\!]_N$ if $v_n = 0$ and into $C'_n = [\![c_n = v_n - 1]\!]_N$ otherwise.

   Consider the case $c_n = 0$. Then, $[\![c_n = 0]\!]_N \equiv \mathtt{ZERO}_n$ and the constraint isz_n is added to the current store. Hence, the guard of the process $\mathbf{when}\ isz_n\ \mathbf{do\ next\ tell}(\mathtt{out}(l_j))$ can be entailed and then, $[\![(li : \mathtt{DECJ}(c_n, l_j, l_k))]\!]_I$ reduces to the process $\mathbf{tell}(idle_n) \parallel \mathbf{next\ tell}(\mathtt{out}(l_j))$. Furthermore, the process $\mathbf{when}\ \textit{not-zero}_n\ \mathbf{do\ next\ tell}(\mathtt{out}(l_k))$ remains blocked since the constraint $\textit{not-zero}_n$ is not added to the current store. We then have the activation of l_j in the next time unit. Since the constraints $idel_n$ is added to the current store, by Proposition 6.3.3, C_n remains the same in Q.

   Now assume that $c_n > 0$. Then, it must be the case that $[\![c_n = k]\!]_N$ adds the constraint $\textit{not-zero}_n$ into the store. Therefore, the process $\mathbf{tell}(dec_n) \parallel \mathbf{next\ tell}(\mathtt{out}(l_k)))$ in the definition of $[\![(li : \mathtt{DECJ}(c_n, l_j, l_k))]\!]_I$ is executed.

   Now we have to show that $C_n \parallel \mathbf{tell}(dec_n)$ reduces to $[\![c_n = v_n - 1]\!]_N$. We consider two cases: when $k > 1$ and $k = 1$. Assume that $k > 0$ and $P = [\![c_n = k]\!]_N \parallel \mathbf{tell}(dec_n)$. We must have the following derivation

$$
\begin{aligned}
P \;\longrightarrow^{*}\; & (\mathbf{local}\, a_1, a_2, ..., a_k)\,(\\
& \quad !\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n \parallel \\
& \quad \ldots \\
& \quad !\,\mathbf{when}\ \mathtt{out}(a_k)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_{k-1}) \\
& \quad \mathbf{when}\ inc_n\ \mathbf{do\ next}\,(\mathbf{local}\, a_{k+1})\,(\mathtt{NOT\text{-}ZERO}_n(a_{k+1}) \parallel \\
& \qquad\qquad\qquad\qquad !\,\mathbf{when}\ \mathtt{out}(a_{k+1})\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_k)) \parallel \\
& \quad \mathbf{when}\ dec_n\ \mathbf{do\ next\ tell}(\mathtt{out}(a_k)) \parallel \\
& \quad \mathbf{when}\ idle_n\ \mathbf{do\ next}\ \mathtt{NOT\text{-}ZERO}_n(a_k) \parallel \\
& \quad \mathbf{tell}(\mathit{not\text{-}zero}_n) \parallel \mathbf{tell}(dec_n)) \\
\longrightarrow^{*}\; & (\mathbf{local}\, a_1, a_2, ..., a_k)\,(\\
& \quad !\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n \parallel \\
& \quad \ldots \\
& \quad !\,\mathbf{when}\ \mathtt{out}(a_k)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_{k-1}) \\
& \quad \mathbf{when}\ inc_n\ \mathbf{do\ next}\,(\mathbf{local}\, a_{k+1})\,(\mathtt{NOT\text{-}ZERO}_n(a_{k+1}) \parallel \\
& \qquad\qquad\qquad\qquad !\,\mathbf{when}\ \mathtt{out}(a_{k+1})\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_k)) \parallel \\
& \quad \mathbf{next\ tell}(\mathtt{out}(a_k)) \parallel \\
& \quad \mathbf{when}\ idle_n\ \mathbf{do\ next}\ \mathtt{NOT\text{-}ZERO}_n(a_k)) \not\longrightarrow
\end{aligned}
$$

We then have $P \Longrightarrow Q$ where

$$
\begin{aligned}
Q \;\equiv\; & (\mathbf{local}\, a_1, a_2, ..., a_k)\,(\\
& \quad !\,\mathbf{when}\ \mathtt{out}(a_1)\ \mathbf{do}\ \mathtt{ZERO}_n \parallel \\
& \quad !\,\mathbf{when}\ \mathtt{out}(a_2)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_1) \parallel \\
& \quad \ldots \\
& \quad !\,\mathbf{when}\ \mathtt{out}(a_k)\ \mathbf{do}\ \mathtt{NOT\text{-}ZERO}_n(a_{k-1}) \parallel \\
& \quad \mathbf{tell}(\mathtt{out}(a_k)))
\end{aligned}
$$

By proposition 6.3.1 we can show that $Q \sim^o [\![c_n = k-1]\!]_N$. The case $k = 1$ is similar to the previous one by noticing that $\mathbf{tell}(\mathtt{out}(a_1))$ triggers the execution of $\mathtt{ZERO}_n$ and then $[\![c_n = k]\!]_N \parallel \mathbf{tell}(dec_n) \Longrightarrow\!\sim^o \mathtt{ZERO}_n \equiv [\![c_n = 0]\!]_N$.

3. $(l_i, \mathtt{HALT})$: In this case, $\mathbf{tell}(\mathtt{out}(l_i))$ in parallel with the encoding of the set of instructions reduces to the process $\mathbf{tell}(\mathtt{halt}) \parallel \mathbf{next\ tell}(\mathtt{out}(l_i)) \parallel \mathbf{tell}(idle_0 \wedge idle_1)$. By Proposition 6.3.3 we can show that the encoding of the counters does not change when passing to the next time unit and then $P \overset{(\mathtt{true},c)}{\Longrightarrow} P$ and $c \models \mathtt{HALT}$

□

Now we prove the converse of the previous lemma.

Lemma 6.3.2 (Soundness)**.** *Let M be a Minsky machine with instructions $(l_1; L_1), ..., (l_n; L_n)$, $[\![\cdot]\!]_C$ be as in Definition 6.2.3 and (l_i, v_0, v_1) be a configuration. If $[\![(l_i, v_0, v_1)]\!]_C \overset{(\mathtt{true},c)}{\Longrightarrow} P$ then, one of the following holds*

1. *There exists a configuration (l_i', v_0', v_1') such that $(l_i, v_0, v_1) \longrightarrow_M (l_i', v_0', v_1')$ and $P \sim^o [\![(l_i', v_0', v_1')]\!]_C$.*
2. *$(l_i, v_0, v_1) \not\longrightarrow_M$, $P \sim^o [\![(l_i, v_0, v_1)]\!]_C$ and $c \models \mathtt{halt}$.*

Proof. By an analysis on the structure of $[\![(l_i, v_0, v_1)]\!]_C$, one can see that the process $\mathbf{tell}(\mathtt{out}(l_i))$ triggers the execution of the definition of the instruction L_i. The other processes evolve according to the type of the instruction L_i. We then proceed by case analysis of $[\![(l_i : L_i)]\!]_I$.

For (1), if $(l_i, v_0, v_1) \longrightarrow_M (l_i', v_0', v_1')$ then li is an increment or a decrement operation. We analyze both cases:

- $(l_i : \mathtt{INC}(c_n, l_j))$. We then have $v_n' = v_n + 1$ and $v_{1-n}' = v_{1-n}$. By exhibiting the same reductions that in the proof of Lemma 6.3.1 case(1), we have $[\![(l_i, v_0, v_1)]\!]_C \Longrightarrow\!\!\sim^o [\![(l_i, v_0', v_1')]\!]_C$.
- $(l_i : \mathtt{DECJ}(c_n, l_j, l_k))$. We have to consider two cases: (a) $c_n = 0$ and then $(l_i, v_0, v_1) \longrightarrow_M (l_j, v_0, v_1)$; (b) $c_n > 0$ and then $(l_i, v_0, v_1) \longrightarrow_M (l_k, v_0', v_1')$ with $v_n' = v_n - 1$ and $v_{1-n}' = v_{1-n}$. In both cases, we can exhibit the same reductions that in Lemma 6.3.1 case (2) to show the following:

  (a) : $[\![(l_i, v_0, v_1)]\!]_C \Longrightarrow\!\!\sim^o [\![(l_j, v_0, v_1)]\!]_C$.

  (b) : $[\![(l_i, v_0, v_1)]\!]_C \Longrightarrow\!\!\sim^o [\![(l_k, v_0', v_1')]\!]_C$

For (2), if $(l_i, v_0, v_1) \not\longrightarrow_M$ then it must be the case that L_i is a $\mathtt{HALT}$ instruction. By an analysis similar to that of (3) in Lemma 6.3.1 we conclude $[\![(l_i, v_0, v_1)]\!]_C \overset{(\mathtt{true},c)}{\Longrightarrow}\!\!\sim^o [\![(l_i, v_0, v_1)]\!]_C$ where $c \models \mathtt{halt}$.

□

6.3.2 Termination and Computations of the Minsky Machine

We conclude this section by presenting a theorem that follows directly from Lemmas 6.3.1 and 6.3.2. This result proves that computations in the Minsky machines correspond to computations in the `utcc` encoding.

Let us define a process decrementing n times the counter c_0. If it succeeds, it outputs the constraint *yes*:

$$\begin{array}{lcl} Dec_0 & \overset{\mathtt{def}}{=} & \mathbf{when}\ isz_0\ \mathbf{do}\ \mathbf{tell}(yes) \\ Dec_n & \overset{\mathtt{def}}{=} & \mathbf{unless}\ isz_0\ \mathbf{next}\ (\mathbf{tell}(dec_0) \parallel Dec_{n-1}) \end{array}$$

Recall that given a process P and a constraint c, $P \Downarrow^c$ means $P = P_1 \overset{(\mathtt{true},c_1)}{\Longrightarrow} P_2 \overset{(\mathtt{true},c_2)}{\Longrightarrow} \ldots P_i \overset{(\mathtt{true},c_i)}{\Longrightarrow} P_{i+1}$ and $c_i \models c$. The following theorem states that a Minsky machine computes the value n (Definition 6.1.1) if and only if after the encoding halts, one can decrement c_0 exactly n times until telling "yes". More precisely:

Theorem 6.3.1 (Correctness)**.** *Let M be a Minsky machine and $\mathsf{M}[\![\cdot]\!]'$ be as in Definition 6.2.1 but where $ins(l_i, \mathtt{HALT}) = Dec_n \parallel\! !\, \mathbf{tell}(idle_1)$.*

The machine M computes the value n iff $\mathsf{M}[\![M]\!]' \Downarrow^{yes}$

Proof. Since we are assuming that counters start in zero and the first instruction to be executed is l_1, it is easy to see that $\mathsf{M}[\![M]\!] \equiv [\![(l_1, 0, 0)]\!]_C$. If M computes the value n, then there exists a derivation $(l_1, 0, 0) \longrightarrow_M^* (l_j, n, v_1)$ with $(l_j : \mathtt{HALT})$. By Lemmas 6.3.1 and 6.3.2, it is possible if and only if $[\![(l_1, 0, 0)]\!]_C \Longrightarrow\!\!\sim^{*o} [\![(l_j, n, v_1)]\!]_C$. Since l_j is a

HALT instruction, by definition of $\mathsf{M}[\![\cdot]\!]'$ the process Dec_n is executed and we must have the following derivations.

$$\begin{aligned}
[\![(l_j, n, v_1)]\!]_C \parallel Q &\Longrightarrow\!\!\sim^o [\![(l_j, n, v_1)]\!]_C \parallel \mathbf{tell}(dec_0) \parallel Dec_{n-1} \\
&\Longrightarrow\!\!\sim^o [\![(l_j, n-1, v_1)]\!]_C \parallel \mathbf{tell}(dec_0) \parallel Dec_{n-2} \\
&\Longrightarrow\!\!\sim^o \ldots \\
&\Longrightarrow\!\!\sim^o [\![(l_j, 0, v_1)]\!]_C \parallel Dec_0 \xRightarrow{(\mathtt{true}, c)} Q'
\end{aligned}$$

where $c \models yes$ and then $\mathsf{M}[\![M]\!]' \Downarrow^{yes}$ □

As an application of the above result, we can show the undecidability of the output equivalence for well-terminated processes.

Corollary 6.3.1. *Fix the underling constraint system to be monadic first-order logic without equality nor function symbols. Then, the question of whether $P \sim^o Q$, given two well-terminated processes P and Q, is undecidable.*

Proof. Given a Minsky machine M, let us define $\mathsf{M}[\![M]\!]'$ as the encoding $\mathsf{M}[\![M]\!]$ except that $ins(\mathtt{HALT}) = \mathbf{skip}$. Notice that $ins(\mathtt{HALT}) = \mathbf{tell}(\mathtt{halt})$ in Figure 6.2. Clearly, M does not halt if and only if $\mathsf{M}[\![M]\!]' \sim^o \mathsf{M}[\![M]\!]$. □

A more compelling application of our encoding is given in the next section where we prove the undecidability of the monadic fragment of FLTL.

6.4 Undecidability of monadic FLTL

In this section we shall state a new undecidability result for monadic FLTL. We shall prove that the monadic fragment of FLTL without equality nor function symbols is strongly incomplete. We start by recalling some results in [Merz 1992] where it is proven that the above fragment of FLTL is decidable. Then we explain why this apparent contradiction with our result arises.

In [Merz 1992] a FLTL named TLV is studied. The logic we presented in Definition 2.4.1 differs from TLV only in that TLV disallows quantification of flexible variables as well as the past operator. We shall see that quantification over flexible variables is fundamental for our encoding of Minsky machines. We also state in Theorem 6.4.1 that the past-free monadic fragment of the FLTL in Definition 2.4.1 without equality nor function symbols is strongly incomplete. This in contrast with the same TLV fragment which is decidable with respect to validity [Merz 1992].

Because of the above-mentioned difference with TLV we shall use the following notation:

Notation 6.4.1. *Henceforth we use TLV-*`flex` *to denote the past-free fragment of the FLTL presented in Definition 2.4.1, i.e., the set of FLTL formulae without occurrences of the past modality $\ominus$.*

Decidability of monadic TLV. In [Merz 1992] it is proven that the problem of validity of a monadic TLV formula A without equality nor function symbols is decidable. This result is proven the same way as the standard decidability result for classical monadic first-order logic (FOL). Namely, by obtaining the prenex form of the formula, getting rid of quantifiers and then reducing the problem to the decidability of propositional LTL.

This strategy does not work in the case of TLV-`flex`. Basically, it is not possible to move a quantifier binding a flexible variable to obtain the prenex form. To see this, consider for example the formula $F = (x = 42 \wedge \circ x \neq 42)$. If x is a flexible variable, notice

that $\Box\exists x F$ is satisfiable whereas $\exists x \Box F$ is not. Hence, moving quantifiers to the outermost position to get a prenex form does not preserve satisfiability. Notice also that if x is a rigid variable instead, $\Box\exists x F$ and $\exists x \Box F$ are both logically equivalent to `false`.

To prove our undecidability result, we shall reduce the validity problem of a TLV-`flex` formula to the halting problem in Minsky machines. We then first need to represent a Minsky machine as a TLV-`flex` formula. This is done by appealing to the encoding of Minsky machines into `utcc` processes in Definition 6.2.1 and then the encoding of `utcc` processes into TLV-`flex` formulae in Definition 5.1.1.

Proposition 6.4.1. *Let M be a Minsky machine and $P = \mathsf{M}[\![M]\!]$ as in Definition 6.2.1. Let $A = \mathsf{TL}[\![P]\!]$ be the FLTL formula obtained as in Definition 5.1.1. Then A is a monadic TLV-*`flex` *formula without equality nor function symbols.*

Proof. Directly from the fact that the constraint system required in $\mathsf{M}[\![\cdot]\!]$ is the monadic fragment without equality nor function symbols of FOL. □

To see the importance of quantifying over flexible variables in the encoding $A = \mathsf{TL}[\![P]\!]$ take the output of x (i.e., the process $\mathbf{tell}(\mathtt{out}(x))$) in the definition of $\mathtt{NOT\text{-}ZERO}_n(x)$ (Figure 6.2). Assume that the variables were rigid. Notice that the abstraction modeling the definition of this procedure is replicated (see Notation 3.3.2). This thus corresponds to a formula of the form $\Box\forall_x \mathtt{out}(x) \Rightarrow F$. Once the formula $\mathtt{out}(a)$ is true, by the rigidity of a, the formula $F[a/x]$ must be true in the following states, which does not correspond to the intended meaning of the machine execution. Instead, if a is flexible, the fact that $\mathtt{out}(a)$ is true at certain state does not imply that $F[a/x]$ must be true in the subsequent states.

Now, using the above proposition and our construction of Minsky machines we have the following:

Lemma 6.4.1. *Given a Minsky machine M, it is possible to construct a monadic TLV-*`flex` *formula without equality nor function symbols F_M such that F_M is valid iff M loops (i.e., it never halts).*

Proof. Let M be a Minsky machine and $P = \mathsf{M}[\![M]\!]''$ where $\mathsf{M}[\![M]\!]''$ is defined as the encoding $\mathsf{M}[\![M]\!]$ in Definition 6.2.1 except that $ins(\cdot)$ (Figure 6.2) adds $\mathbf{tell}(\mathtt{running})$ in parallel to the encoding of all instructions but `HALT`, i,e,:

$$\begin{array}{lcl} ins''(l_i, \mathtt{HALT}) & = & ins(l_i, \mathtt{HALT}) \\ ins''(l_i, \mathtt{INC}(c_n, l_j)) & = & ins(l_i, \mathtt{INC}(c_n, l_j)) \parallel \mathbf{tell}(\mathtt{running}) \\ ins''(l_i, \mathtt{DECJ}(c_n, l_j, l_k)) & = & ins(l_i, \mathtt{DECJ}(c_n, l_j, l_k)) \parallel \mathbf{tell}(\mathtt{running}) \end{array}$$

Take $A = \mathsf{TL}[\![P]\!]$. One can verify that if $A \models_T \Diamond \mathtt{halt}$ then, it must be the case that there exists $j \geq 0$ such that $Cut_F(A, j) \models_T \Diamond \mathtt{halt}$. Let $F_M = A \Rightarrow \Box \mathtt{running}$. One can show that F_M is not valid if and only if $Cut_F(A, j) \models_T \Diamond \mathtt{halt}$ since when `halt` can be deduced, by construction of ins'', `running` cannot be deduced. By Corollary 5.2.2, $Cut_F(A, j) \models_T \Diamond \mathtt{halt}$ iff $P \Downarrow^{\mathtt{halt}}$ and from Lemmas 6.3.1 and 6.3.2, $P \Downarrow^{\mathtt{halt}}$ iff M halts. Therefore, F_M is valid iff M never halts. □

Since the set of looping Minsky machines (i.e. the complement of the halting problem) is not recursively enumerable, a finitistic axiomatization of monadic TLV-`flex` without equality nor function symbols would yield a recursively enumerable set of tautologies.

Theorem 6.4.1 (Incompleteness)**.** *There is no a sound and complete finitistic axiomatization for monadic TLV-*`flex` *without equality nor function symbols.*

Proof. Directly from Lemma 6.4.1. □

From this corollary it follows that the *validity problem* in the above-mentioned monadic fragment of TLV-`flex` is undecidable. Our results then show that the restriction on the quantification of flexible variables is necessary for the decidability result of monadic TLV in [Merz 1992].

In Appendix A we present an alternative proof of the Theorem 6.4.1 using only argument from logic. We prove that the formula corresponding to the process $\mathsf{M}[\![M]\!]$ faithfully describe the behavior of the machine M. Then we show that there exists a formula that is valid if and only if M never halts.

6.5 Encoding the λ-calculus into utcc

In this section we give a *compositional* encoding of the call-by-name λ-calculus into `utcc` processes. This encoding is a significant application showing how `utcc` is able to mimic one of the most notable and simple computational models achieving Turing completeness. Here, the ability to express mobility is central to our encoding that is built upon the ideas in the encoding of the λ-calculus into the π-calculus in [Milner 1992b, Sangiorgi 1992].

We shall recall briefly some notions of the lazy λ-calculus [Abramsky 1993] and the encoding of it into the π-calculus [Milner 1992b, Sangiorgi 1992].

6.5.1 The call-by-name λ-calculus

Terms in the λ-calculus denoted by $M, N, ...$ are built from variables $x, y, ...$ by the following syntax:

$$M ::= x \;\mid\; (\lambda x.M) \;\mid\; (MN)$$

The term $(\lambda x.M)$ is known as λ-abstraction and (MN) as the *application* of M to N.

Computations in the *call-by-name* λ-calculus are described by the relation $\longrightarrow_\lambda$:

$$\beta\ \frac{}{((\lambda x.M)N) \longrightarrow_\lambda M[N/x]} \qquad \mu\ \frac{M \longrightarrow_\lambda M'}{MN \longrightarrow_\lambda M'N}$$

Rule β replaces the placeholder x by the argument N in the body M. The parameter N in such an expression is *not evaluated* before the substitution takes place. The expression $(\lambda x.M)$ is called β-redex and the result of the reduction, $M[N/x]$, is called *contractum*. Rule μ allows us to replace the leftmost β-redex in the *application* (MN) by its contractum. Notice that terms in the body of an abstraction are not reduced.

Since the parameters are not evaluated, the call-by-name strategy allows for the manipulation of infinite objects. Furthermore, reductions are *deterministic*: the redex is always at the extreme left of the term [Abramsky 1993].

6.5.2 Encoding the λ-calculus into the π-calculus

The encoding of the λ-calculus in [Milner 1992b] and [Sangiorgi 1992] makes use of channels to represent the linkage between an abstraction of the form $M = \lambda x.M_0$ and the sequence of arguments $M_1M_2...$, when applying M to $M_1M_2....$ The encoding of M is then parametric to a channel name indicating where to find the first argument. This way, function application is a particular form of parallel combination of two agents, the function and its argument. Beta-reduction is modeled as process interaction. Since channels can communicate only names, the communication of a term is simulated by the communication of a trigger for it [Sangiorgi 1998].

The translation is inductively defined as follows

$$\begin{array}{lll} \text{variable} & [\![x]\!]^{\pi}_{u} & \stackrel{\text{def}}{=} \ \overline{x}\langle u\rangle \\ \lambda\text{-abstraction} & [\![\lambda x.M]\!]^{\pi}_{u} & \stackrel{\text{def}}{=} \ u(x,v).[\![M]\!]^{\pi}_{v} \\ \text{Application} & [\![(MN)]\!]^{\pi}_{u} & \stackrel{\text{def}}{=} \ \nu v([\![M]\!]^{\pi}_{v} \mid \nu w(\overline{v}\langle w,u\rangle \mid !\, w(m).[\![N]\!]^{\pi}_{m})) \end{array}$$

In the encoding above, the application MN causes that $[\![M]\!]^{\pi}_{v}$ consumes the process $\overline{v}\langle x,v'\rangle$, thus finding the name x of the first argument and a link (v') to the reminder of the arguments.

See [Sangiorgi 1992] and [Sangiorgi 1998] for further details on this encoding and the correspondence theorems.

6.5.3 Encoding the λ-calculus into utcc

In our encoding we shall mimic the input and output actions in the π-calculus encoding of the λ-calculus. Notice that inputs and outputs in the π-calculus disappear only after being involved in an input-output interaction. In utcc, the tell and abstraction processes can be thought of as being outputs and inputs, respectively, in π, but they are not automatically transferred from one time unit to the next one—intuitively, they will disappear right after the current time unit even if they did not interact.

Consequently, to mimic π inputs we define the derived operator $(\mathbf{wait}\ \vec{x}; c)\ \mathbf{do}\ Q$ that *waits*, possibly for several time units until for some $\vec{t}$, $c[\vec{t}/\vec{x}]$ holds. Then it executes $Q[\vec{t}/\vec{x}]$.

Definition 6.5.1 (Wait Process)**.** *Assuming an uninterpreted predicate* $\mathtt{out}'$ *in the signature of the constraint system and* $stop, go \notin fv(P)$ *we define*

$$\begin{array}{rcl} (\mathbf{wait}\ \vec{x}; c)\ \mathbf{do}\ P & \stackrel{\text{def}}{=} & (\mathbf{local}\, stop, go)\, \mathbf{tell}(\mathtt{out}'(go)) \\ & & \quad \|!\, \mathbf{unless}\ \ \mathtt{out}'(stop)\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go)) \\ & & \quad \|!\, (\mathbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go))\, (P \,\|!\, \mathbf{tell}(\mathtt{out}'(stop))) \end{array}$$

In the previous definition, the guard of the abstraction c is augmented with the constraint $\mathtt{out}'(go)$ which is added to the store until the constraint $\mathtt{out}'(stop)$ is deduced. The latter is added once the body of the abstraction (i.e., P) is executed.

Notation 6.5.1. *Recall that the empty sequence of terms is written as* ε*. We shall use* $\mathbf{whenever}\ c\ \mathbf{do}\ P$ *as a shorthand for* $(\mathbf{wait}\ \varepsilon; c)\ \mathbf{do}\ P$*.*

To mimic π outputs, we require a derived constructor able to perform the output until some process is able to *"read"* the constraint produced—after interacting with an input process. We shall write $\underline{\mathbf{tell}}(c)$ for the persistent output of c until some process reads c. We also define an auxiliary input process that adds a constraint acknowledging the reading of c namely $(\underline{\mathbf{wait}}\ \vec{x}; c)\ \mathbf{do}\ P$.

Definition 6.5.2 (Persistent Tells and Waits)**.** *Assuming* $stop, go \notin fv(c)$*. Define*

$$\begin{array}{rcl} \underline{\mathbf{tell}}(c) & \stackrel{\text{def}}{=} & (\mathbf{local}\, go, stop)\, \mathbf{tell}(\mathtt{out}'(go)) \,\|!\, \mathbf{when}\ \ \mathtt{out}'(go)\ \mathbf{do}\ \mathbf{tell}(c) \,\| \\ & & \quad !\, \mathbf{unless}\ \ \mathtt{out}'(stop)\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go)) \,\| \\ & & \quad !\, \mathbf{when}\ \bar{c}\ \mathbf{do}\ !\, \mathbf{tell}(\mathtt{out}'(stop)) \\ (\underline{\mathbf{wait}}\ \vec{x}; c)\ \mathbf{do}\ P & \stackrel{\text{def}}{=} & (\mathbf{wait}\ \vec{x}; c)\ \mathbf{do}\ (P \parallel \mathbf{tell}(\bar{c})) \end{array}$$

Intuitively, the constraint $\bar{c}$ is added to the store in order to acknowledge that c was read. Once this happens, the resulting process R is only able to output constraints of the form $\mathtt{out}'(x)$ where x is a local variable. This is due to the processes of the form $!\, \mathbf{tell}(\mathtt{out}'(stop))$ in R. We shall elaborate more about this issue in the next section where we prove that R can be viewed as an *inactive* process for the execution of the encoding.

The Encoding. The encoding below maps arbitrary λ-terms into utcc processes. We presuppose a constraint system with two uninterpreted predicates $\mathtt{out}_2$ and $\mathtt{out}_3$ and the corresponding acknowledgments $\overline{\mathtt{out}}_2$ and $\overline{\mathtt{out}}_3$. For the sake of simplicity, we shall omit the sub-indexes in $\mathtt{out}_2$ and $\mathtt{out}_3$ and they are understood as the arity of $\mathtt{out}$.

$$
\begin{array}{lllll}
\text{variable} & [\![x]\!]^u_\lambda & \stackrel{\text{def}}{=} & \underline{\mathbf{tell}}(\mathtt{out}(x,u)) \\
\lambda\text{-abstraction} & [\![\lambda x.M]\!]_u & \stackrel{\text{def}}{=} & (\underline{\mathbf{wait}}\ x,v;\mathtt{out}(u,x,v))\ \mathbf{do}\ \mathbf{next}\ [\![M]\!]^v_\lambda \\
\text{Application} & [\![(MN)]\!]^u_\lambda & \stackrel{\text{def}}{=} & (\mathbf{local}\ v)\,([\![M]\!]_v \parallel \\
 & & & \qquad (\mathbf{local}\ w)\ \underline{\mathbf{tell}}(\mathtt{out}(v,w,u) \parallel \\
 & & & \qquad\qquad !\,(\underline{\mathbf{wait}}\ m;\mathtt{out}(w,m))\ \mathbf{do}\ \mathbf{next}\ [\![N]\!]_m))
\end{array}
$$

As the reader can see, our encoding follows directly from that of the λ-calculus into the π-calculus. Intuitively, the constraint $\mathtt{out}_2(a,x)$ represents sending the name x on channel a as in Example 3.6.2. Similarly, $\mathtt{out}_3(a,x,y)$ represents sending both x and y on a.

6.5.4 Correctness of the Encoding

In this section we prove the correspondence between the derivations of a lambda term and the derivations of its corresponding utcc process. Before doing that, let us introduce some facts on the processes **tell** and **wait**.

Notice that once the processes **tell** and **wait** interact, their continuation in the next time unit is a process able to output only a constraint of the form $\exists_x\,\mathtt{out}'(x)$. We then define the following equivalence relation that allows us to "ignore" these processes.

Definition 6.5.3 (Observables)**.** *Let $\sim^o$ be the output equivalent relation in Definition 3.7.1. We say that P and Q are observable equivalent, notation $P \sim^{obs} Q$, if*

$$P \parallel !\,\mathbf{tell}(\exists_x \mathtt{out}'(x)) \sim^o Q \parallel !\,\mathbf{tell}(\exists_x \mathtt{out}'(x))$$

Using the previous equivalence relation, we can show the following.

Observation 6.5.1. *Assume that $c(\vec{x})$ is a predicate symbol of arity $|\vec{x}|$.*

1. *If* $\mathtt{true} \not\models c$ *then* $(\underline{\mathbf{wait}}\ \vec{x};c)\ \mathbf{do}\ P \Longrightarrow\!\!\!\!\!\!\rightarrow (\underline{\mathbf{wait}}\ \vec{x};c)\ \mathbf{do}\ P$.
2. *If* $P \equiv \underline{\mathbf{tell}}(c(\vec{t})) \parallel (\underline{\mathbf{wait}}\ \vec{x};c(\vec{x}))\ \mathbf{do}\ \mathbf{next}\, Q$ *then* $P \Longrightarrow\!\!\!\!\!\!\rightarrow \sim^{obs} Q[\vec{t}/\vec{x}]$.

Proof. For (1), one can show that there is a derivation of the form

$$
\begin{array}{lll}
(\underline{\mathbf{wait}}\ \vec{x};c)\ \mathbf{do}\ P & \longrightarrow^* & (\mathbf{local}\ stop, go; \mathtt{out}'(go)) \\
 & & \quad \parallel \mathbf{unless}\ \ \mathtt{out}'(stop)\ \mathbf{next}\,\mathbf{tell}(\mathtt{out}'(go)) \\
 & & \quad \parallel \mathbf{next}\,!\,\mathbf{unless}\ \ \mathtt{out}'(stop)\ \mathbf{next}\,\mathbf{tell}(\mathtt{out}'(go)) \\
 & & \quad \parallel (\mathbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go))\,(P \parallel \mathbf{tell}(\overline{c}) \parallel !\,\mathbf{tell}(\mathtt{out}'(stop)) \\
 & & \quad \parallel \mathbf{next}\,!\,(\mathbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go))\,(P \parallel \mathbf{tell}(\overline{c}) \parallel !\,\mathbf{tell}(\mathtt{out}'(stop)) \not\longrightarrow
\end{array}
$$

Notice that the **unless** process above executes the process $\mathbf{tell}(\mathtt{out}'(go))$ in the next time unit. By observing the definition of **wait**, it is easy to see that $(\underline{\mathbf{wait}}\ \vec{x};c)\ \mathbf{do}\ P \Longrightarrow\!\!\!\!\!\!\rightarrow (\underline{\mathbf{wait}}\ \vec{x};c)\ \mathbf{do}\ P$.

For (2), assume that $P = \mathbf{next}\, Q$ and let $R = \underline{\mathbf{tell}}(c(\vec{t})) \parallel (\underline{\mathbf{wait}}\ \vec{x}; c(\vec{x}))\ \mathbf{do}\ P$. One can show that there is a derivation of the form

$$
\begin{array}{rll}
R & \longrightarrow^* & (\mathbf{local}\, go, stop; \mathtt{out}'(go))\, \mathbf{when}\ \mathtt{out}'(go)\ \mathbf{do}\ \mathbf{tell}(c(\vec{t})) \parallel \\
& & \qquad\qquad !\, \mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go)) \parallel \\
& & \qquad\qquad !\, \mathbf{when}\ \overline{c}(\vec{t})\ \mathbf{do}\ !\, \mathbf{tell}(\mathtt{out}'(stop)) \parallel \\
& & (\mathbf{local}\, stop', go'; \mathtt{out}'(go')) \\
& & \quad \parallel\! !\, \mathbf{unless}\ \mathtt{out}'(stop')\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go')) \\
& & \quad \parallel\! !\, (\mathbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go'))\, (P \parallel \mathbf{tell}(\overline{c}(\vec{t})) \parallel\! !\, \mathbf{tell}(\mathtt{out}'(stop')) \\
& \longrightarrow^* & (\mathbf{local}\, go, stop; \mathtt{out}'(go) \wedge c(\vec{t}))\ \parallel \\
& & \qquad\qquad !\, \mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go)) \parallel \\
& & \qquad\qquad !\, \mathbf{when}\ \overline{c}(\vec{t})\ \mathbf{do}\ !\, \mathbf{tell}(\mathtt{out}'(stop)) \parallel \\
& & (\mathbf{local}\, stop', go'; \mathtt{out}'(go')) \\
& & \quad \parallel\! !\, \mathbf{unless}\ \mathtt{out}'(stop')\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go')) \\
& & \quad \parallel\! !\, (\mathbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go'))\, (P \parallel \mathbf{tell}(\overline{c}(\vec{t})) \parallel\! !\, \mathbf{tell}(\mathtt{out}'(stop'))
\end{array}
$$

Since $stop, go, stop', go' \notin fv(c) \cup fv(P)$ we must have

$$
\begin{array}{rll}
R & \longrightarrow^* & P[\vec{t}/\vec{x}] \parallel (\mathbf{local}\, go, stop; \mathtt{out}'(go) \wedge c(\vec{t}) \wedge \mathtt{out}'(stop))\, \mathbf{next}\, !\, \mathbf{tell}(\mathtt{out}'(stop)) \parallel \\
& & \qquad\qquad \mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go)) \parallel \\
& & \qquad\qquad \mathbf{next}\, !\, \mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go)) \parallel \\
& & (\mathbf{local}\, stop', go'; \mathtt{out}'(go') \wedge \overline{c}(\vec{t}) \wedge \mathtt{out}'(stop'))\, \mathbf{next}\, !\, \mathbf{tell}(\mathtt{out}'(stop')) \\
& & \quad \parallel \mathbf{unless}\ \mathtt{out}'(stop')\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go')) \\
& & \quad \parallel \mathbf{next}\, !\, \mathbf{unless}\ \mathtt{out}'(stop')\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go')) \\
& & \quad \parallel (\mathbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go') \wedge \vec{x} \neq \vec{t})\, (P \parallel \mathbf{tell}(\overline{c}(\vec{t})) \parallel\! !\, \mathbf{tell}(\mathtt{out}'(stop')) \\
& & \quad \parallel \mathbf{next}\, !\, (\mathbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go'))\, (P \parallel \mathbf{tell}(\overline{c}(\vec{t})) \parallel\! !\, \mathbf{tell}(\mathtt{out}'(stop')) \\
& \longrightarrow^* & P[\vec{t}/\vec{x}] \parallel (\mathbf{local}\, go, stop; \mathtt{out}'(go) \wedge c(\vec{t}) \wedge \mathtt{out}'(stop))\, \mathbf{next}\, !\, \mathbf{tell}(\mathtt{out}'(stop)) \parallel \\
& & \qquad\qquad \mathbf{next}\, !\, \mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go)) \parallel \\
& & (\mathbf{local}\, stop', go'; \mathtt{out}'(go') \wedge \overline{c}(\vec{t}) \wedge \mathtt{out}'(stop'))\, \mathbf{next}\, !\, \mathbf{tell}(\mathtt{out}'(stop')) \\
& & \quad \parallel \mathbf{next}\, !\, \mathbf{unless}\ \mathtt{out}'(stop')\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go')) \\
& & \quad \parallel (\mathbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go') \wedge \vec{x} \neq \vec{t})\, (P \parallel \mathbf{tell}(\overline{c}(\vec{t})) \parallel\! !\, \mathbf{tell}(\mathtt{out}'(stop')) \\
& & \quad \parallel \mathbf{next}\, !\, (\mathbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go'))\, (P \parallel \mathbf{tell}(\overline{c}(\vec{t})) \parallel\! !\, \mathbf{tell}(\mathtt{out}'(stop'))) \not\longrightarrow
\end{array}
$$

Let $R' = P[\vec{t}/\vec{x}] \parallel R''$ be the process in the configuration above. Notice that in R'' the processes $\mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go))$ and $\mathbf{unless}\ \mathtt{out}'(stop')\ \mathbf{next}\, \mathbf{tell}(\mathtt{out}'(go'))$ cannot add the constraints $\mathtt{out}'(go)$ and $\mathtt{out}'(go')$ because of the processes $!\, \mathbf{tell}(\mathtt{out}'(stop))$ and $!\, \mathbf{tell}(\mathtt{out}'(stop'))$. Hence, the process R'' can only output constraints of the form $\mathtt{out}'(x)$ where x is a local variable and P cannot be spawn from R''. Since $P = \mathbf{next}\, Q$ we conclude $R \Longrightarrow\!\!\sim^{obs} Q[\vec{t}/\vec{x}]$. □

For the sake of presentation, the notation below introduces a shorthand for the residual process generated by a process of the form $\underline{\mathbf{tell}}(c(\vec{t})) \parallel (\underline{\mathbf{wait}}\ \vec{x}; c)\ \mathbf{do}\ Q$.

Notation 6.5.2 (Residual process)**.** *Let $c(\vec{x})$ be a predicate symbol of arity $|\vec{x}|$ and $P = \underline{\mathbf{tell}}(c(\vec{t})) \parallel (\underline{\mathbf{wait}}\ \vec{x}; c)\ \mathbf{do}\ Q$. We shall use $\mathtt{inact\text{-}wait}(\vec{x}, \vec{t}, c, Q)$ to denote the process $P' \parallel P''$ where*

$$
\begin{array}{rcl}
P' & \equiv & (\textbf{local}\, go, stop; \mathtt{out}'(go) \wedge \mathtt{out}'(stop) \wedge c(\vec{t})) \\
& & \qquad \textbf{next}\,!\,\textbf{unless}\ \mathtt{out}'(stop)\ \textbf{next}\,\textbf{tell}(\mathtt{out}'(go)) \parallel \\
& & \qquad \textbf{next}\,!\,\textbf{tell}(\mathtt{out}'(stop)) \\
P'' & \equiv & (\textbf{local}\, stop', go'; \mathtt{out}'(go') \wedge \overline{c}(\vec{t}) \wedge \mathtt{out}'(stop'))\, \textbf{next}\,!\,\textbf{tell}(\mathtt{out}'(stop')) \\
& & \quad \parallel \textbf{next}\,!\,\textbf{unless}\ \mathtt{out}'(stop')\ \textbf{next}\,\textbf{tell}(\mathtt{out}'(go')) \\
& & \quad \parallel (\textbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go') \wedge \vec{x} \neq \vec{t})\, (Q \parallel \textbf{tell}(\overline{c}(\vec{t})) \parallel !\, \textbf{tell}(\mathtt{out}'(stop')) \\
& & \quad \parallel \textbf{next}\,!\,(\textbf{abs}\ \vec{x}; c \wedge \mathtt{out}'(go'))\, (Q \parallel \textbf{tell}(\overline{c}(\vec{t})) \parallel !\, \textbf{tell}(\mathtt{out}'(stop'))
\end{array}
$$

Now we are ready to state the correctness of our encoding.

Theorem 6.5.1 (Correctness)**.** *Let M and N be λ terms.*

- *(*Soundness*). If $M \longrightarrow_\lambda N$, there is P s.t. $[\![M]\!]_u \Longrightarrow^* P$ and $P \sim^{obs} [\![N]\!]_u$.*
- *(*Completeness*). If $[\![M]\!]_u \Longrightarrow^* P$, there is N' s.t $M \longrightarrow_\lambda^* N'$ and $[\![N']\!]_u \sim^{obs} P$.*

Proof. The proof follows that in [Sangiorgi 1992] for the encoding of the lazy-lambda calculus (call-by-name lambda calculus) into HOπ (a higher-order extension of π).

Both, soundness and completeness are proven by induction on the structure of M. We consider three cases:

1. $M = \lambda x.M'$. In this case, M does not exhibit any reduction. By definition of $[\![\cdot]\!]_\lambda$ we have
$$[\![\lambda x.M']\!]_\lambda^u = (\underline{\textbf{wait}}\ x, v; \mathtt{out}(u, x, v))\ \textbf{do next}\ [\![M']\!]_\lambda^v$$
By (1) in Observation 6.5.1 we conclude
$$[\![\lambda x.M']\!]_\lambda^u \Longrightarrow \sim^{obs} [\![\lambda x.M']\!]_\lambda^u$$

2. $M = M_1 M_2$ and $M_1 = \lambda x.M_3$. We then have $M \longrightarrow_\lambda M_3[M_2/x]$. According to the definition of $[\![\cdot]\!]_\lambda$ we have the following processes
$$
\begin{array}{rcl}
[\![M]\!]_\lambda^u & = & (\textbf{local}\, v)\, ([\![M_1]\!]_\lambda^v \parallel (\textbf{local}\, w)\, \underline{\textbf{tell}}(\mathtt{out}(v, w, u) \parallel \\
& & \qquad\qquad !\, (\underline{\textbf{wait}}\ m; \mathtt{out}(w, m))\ \textbf{do}\ [\![M_2]\!]_\lambda^m)) \\
[\![M_1]\!]_\lambda^v & = & (\underline{\textbf{wait}}\ x, v'; \mathtt{out}(v, x, v'))\ \textbf{do next}\ [\![M_3]\!]_\lambda^{v'}
\end{array}
$$
Observe that the process $\underline{\textbf{tell}}(\mathtt{out}(v, w, u))$ in the definition of $[\![M]\!]_\lambda^u$ interacts with the $\underline{\textbf{wait}}$ process in the definition of $[\![M_1]\!]_\lambda^v$ and we obtain the following derivation
$$
\begin{array}{rcl}
[\![M]\!]_\lambda^u & \longrightarrow^* & (\textbf{local}\, v, w)\, (\textbf{next}\ [\![M_3]\!]_\lambda^{v'}[w/x, u/v'] \parallel \\
& & \qquad \mathtt{inact\text{-}wait}(xv', wu, \mathtt{out}(v, x, v'), \textbf{next}\ [\![M_2]\!]_\lambda^m) \parallel \\
& & \qquad \textbf{next}\,!\,(\underline{\textbf{wait}}\ m; \mathtt{out}(w, m))\ \textbf{do next}\ [\![M_2]\!]_\lambda^m) \not\longrightarrow
\end{array}
$$
Since v' is substituted by u we can prove that
$$[\![M]\!]_\lambda^u \Longrightarrow \sim^{obs} (\textbf{local}\, v, w)\, ([\![M_3]\!]_\lambda^u[w/x] \parallel !\, (\underline{\textbf{wait}}\ m; \mathtt{out}(w, m))\ \textbf{do next}\ [\![M_2]\!]_\lambda^m)$$
Now we shall prove that any application of x to a term M' in $M_3[M_2/x]$ must reduce to the application of M_2 to M'. Without loss of generality, assume that $M_3 = M_3' M'$ and $M_3' \longrightarrow_\lambda^* x$. Then, it must be the case that $M_3[M_2/x] \longrightarrow_\lambda^* M_2 M'[M_2/x]$. By inductive hypothesis we have
$$[\![M]\!]_\lambda^u \Longrightarrow^* \sim^{obs} P \equiv (\textbf{local}\, v, w)\, ([\![xM']\!]_\lambda^u[w/x] \parallel !\, (\underline{\textbf{wait}}\ m; \mathtt{out}(w, m))\ \textbf{do}\ [\![M_2]\!]_\lambda^m)$$

By the definition of $[\![\cdot]\!]_\lambda$ we obtain

$$P \;=\; (\mathbf{local}\, v, w, v')\,([\![x]\!]_\lambda^{v'}[w/x] \parallel !\,(\underline{\mathbf{wait}}\ m; \mathtt{out}(w, m))\ \mathbf{do\ next}\ [\![M_2]\!]_\lambda^m \parallel (\mathbf{local}\, w')\,(\underline{\mathbf{tell}}(\mathtt{out}\, v', w', u) \parallel !\,(\underline{\mathbf{wait}}\ m'; \mathtt{out}(w', m'))\ \mathbf{do\ next}\ [\![M']\!]_\lambda^{m'}[w/x]))$$

Notice that $[\![x]\!]_\lambda^{v'}[w/x] = \underline{\mathbf{tell}}(w, v')$. This way, the process

$$(\underline{\mathbf{wait}}\ m; \mathtt{out}(w, m))\ \mathbf{do\ next}\ [\![M_2]\!]_\lambda^m$$

triggers the execution of $[\![M_2]\!]_\lambda^m[v'/m]$ and we derive

$$P \;\Longrightarrow\!\!\!\!\!\sim^{obs}\; (\mathbf{local}\, v, w, v')\,([\![M_2]\!]_\lambda^{v'} \parallel !\,(\underline{\mathbf{wait}}\ m; \mathtt{out}(w, m))\ \mathbf{do\ next}\ [\![M_2]\!]_\lambda^m \parallel (\mathbf{local}\, w')\,(\underline{\mathbf{tell}}(\mathtt{out}\, v', w', u) \parallel !\,(\underline{\mathbf{wait}}\ m'; \mathtt{out}(w', m'))\ \mathbf{do\ next}\ [\![M']\!]_\lambda^{m'}[w/x]))$$

We then conclude $[\![M]\!]_\lambda^u \;\Longrightarrow\!\!\!\!\!\sim^{*obs}\; [\![M_2M'[M_2/x]]\!]_\lambda^u$

3. $M = M_1M_2$ and M_1 is not of the form $\lambda x.M_3$. It must be the case that $M_1 \longrightarrow_\lambda M_1'$ and then $M \longrightarrow_\lambda M_1'M_2$. By induction we have

$$[\![M_1]\!]_\lambda^v \;\Longrightarrow\!\!\!\!\!\sim^{*obs}\; [\![M_1']\!]_\lambda^v$$

By the definition of $[\![\cdot]\!]_\lambda$ we have

$$[\![M]\!]_\lambda^u \;=\; (\mathbf{local}\, v)\,([\![M_1]\!]_v \parallel (\mathbf{local}\, w)\,\underline{\mathbf{tell}}(\mathtt{out}(v, w, u) \parallel !\,(\underline{\mathbf{wait}}\ m; \mathtt{out}(w, m))\ \mathbf{do\ next}\ [\![M_2]\!]_m))$$

Since M_1 is not of the form $\lambda x.M_3$, the constraint $\mathtt{out}(v, w, u)$ cannot interact with any process in $[\![M_1]\!]_\lambda^v$ (notice that only the encoding of λ-abstractions can synchronize with ternary predicates). Furthermore, since the variable w is local in $(\underline{\mathbf{wait}}\ m; \mathtt{out}\, w, m)$ **do** $[\![M_2]\!]_\lambda^m$, the process $[\![M_1]\!]_\lambda^v$ cannot add a constraint enabling the guard of this **wait** process. Then, the only derivation of $[\![M]\!]_\lambda^v$ is the following

$$[\![M]\!]_\lambda^u \;\Longrightarrow\!\!\!\!\!\sim^{obs}\; (\mathbf{local}\, v)\,([\![M_1']\!]_\lambda^v \parallel (\mathbf{local}\, w)\,\underline{\mathbf{tell}}(\mathtt{out}(v, w, u) \parallel !\,(\underline{\mathbf{wait}}\ m; \mathtt{out}(w, m))\ \mathbf{do\ next}\ [\![M_2]\!]_\lambda^m))$$

We then conclude $[\![M]\!]_\lambda^u \;\Longrightarrow\!\!\!\!\!\sim^{obs}\; [\![M_1'M2]\!]_\lambda^u$

□

6.6 Summary and Related Work

In this chapter we studied the expressiveness of `utcc`. We showed that well-terminated processes and a very simple constraint system are enough to encode Turing-powerful formalisms. More precisely, using the monadic fragment of first-order logic (FOL) without equality nor function symbols, we encoded Minsky machines. Furthermore, using a polyadic constraint system, we proposed a compositional encoding of the call-by-name λ-calculus into `utcc` following the ideas in [Milner 1992b, Sangiorgi 1998].

As an application of this expressiveness study, we showed that the monadic fragment without equality nor function symbols of FLTL is strongly incomplete. This result refutes a decidability conjecture for FLTL in [Valencia 2005]. It also justifies the restriction imposed in previous decidability results on the quantification of flexible-variables [Merz 1992]. This dissertation then fills a gap on the decidability study of monadic FLTL.

The material of this chapter was originally published as [Olarte 2008b].

Related Work. The expressivity of CCP calculi has been explored in [Saraswat 1994, Nielsen 2002b, Valencia 2005]. These works show that `tcc` processes are finite-state. The results in [Nielsen 2002b] also imply that the processes of the extension of `tcc` with arbitrary recursive definitions are not finite-state. Nevertheless these results do not imply that they can encode Turing-expressive formalisms.

There are several works addressing the decidability of fragments of FLTL. The work in [Merz 1992] shows that the validity problem in monadic TLV without equality nor function symbols is decidable. As mentioned before, TLV unlike TLV-`flex` does not allow quantification over flexible variables. Our decidability results justifies the imposition of this quantification restriction.

The work in [Valencia 2005] shows the decidability of the satisfiability problem of the negation-free fragment of TLV-`flex`. It was also suggested in [Valencia 2005] that one could dispense with the restriction of negation-free. The Theorem 6.4.1 refutes this since including negation one can obviously define universal quantification (not present in the negation-free fragment of [Valencia 2005]) and then be able to reproduce the encoding of looping Minsky machines here presented.

The full expressiveness of process calculi such as CCS [Milner 1992a] and the π-calculus [Milner 1999, Sangiorgi 2001] (or fragments of them) has been also proven by exhibiting encodings of Register machines (or Minsky machines) into the target language. The encoding we presented in Section 6.2 was inspired on the ideas of [Busi 2003, Busi 2004, Palamidessi 2006], where a sequence of local variables is used to represent a number. Using these encodings, the works in [Busi 2003, Busi 2004, Aranda 2009] compare the expressiveness power of different syntactic variants of CCS. These results allowed the authors to prove the (un)decidability of the problem whether two processes are equivalent under a given equivalence relation or to know if a process can exhibit terminating computations (convergence). Similarly, in [Palamidessi 2006], it is shown that the persistent fragment of the π-calculus (all inputs and outputs are replicated) is enough to encode Minsky machines. This result along with a characterization of this fragment into First-Order Logic allowed the authors to identify decidable classes with respect to barbed (output) reachability.

Expressiveness of FLTL. Based on the undecidability result of TLV($\emptyset$) [Szalas 1988] (i.e. TLV with the empty set of predicates), [Merz 1992] proves an incompleteness result for monadic without equality and function symbols TLP logic. Unlike TLV, in TLP the interpretation of the predicates is flexible (state dependent) and all the variables are rigid. [Merz 1992] also relates undecidability results of n-adic fragments of TLP with undecidability results of $n+1$-adic fragments of TLV. Thus adding binary predicates turns TLV strongly incomplete.

In [Hodkinson 2000] the *monodic* fragment of FLTL is introduced. A formula is monodic if every subformulae beginning with a temporal operator have at most one free variable. In this case the authors use a TLP-like semantics and conclude that the set of valid formulae in the 2-variable monadic fragment (i.e. monadic formulae with at most 2 distinct individual variables) is not recursively enumerable even considering finite domains in the interpretation. Nevertheless validity in the fragment of 2-variable monodic formulae is decidable. In [Degtyarev 2002] these results are extended claiming the undecidability for validity in the monodic monadic 2-variable with equality fragment of TLP. The work in [Hussak 2008] extends the results in [Hodkinson 2000] by showing the decidability of the monadic, monodic fragment of TLP with function symbols.

Finally, our encoding in `utcc` of the λ-calculus builds on the corresponding encoding in the π-calculus in [Milner 1992b, Sangiorgi 1998] and in Higher-order Linear CCP (HL-CCP)

[Saraswat 1992]. HL-CCP as presented in [Saraswat 1992] is a calculus closely related to the π-calculus and, unlike `utcc`, is higher-order. The encoding in [Saraswat 1992] appeals to the higher-order nature of HL-CCP.

Chapter 7

Denotational Semantics

In Chapter 4 we studied a symbolic semantics for `utcc` aiming at observing the behavior of non well-terminated processes. We showed that this semantics gives a compact (and abstract) representation of the outputs of a process by using temporal formulae. In the case of the abstraction operator $P = (\mathbf{abs}\ \vec{x}; c)\, Q$, this semantics differs from the operational semantics in that symbolically, the current store is augmented with a constraint of the form $\forall_{\vec{x}}(c \Rightarrow F)$ regardless if d entails $c[\vec{t}/\vec{x}]$ for some $\vec{t}$ or not. In contrast, we know that P operationally does not exhibit any transition if d does not entail $c[\vec{t}/\vec{x}]$ for some $\vec{t}$.

In this chapter we first characterize the output of a process from the symbolic constraints we observe from it. To do this, we shall define the symbolic input-output behavior of a process as follows. Assume that $P \xrightarrow{(d,e')}_s Q$, i.e., under input d, the process P produces symbolically e'. We shall say that the output of P under input d is e where e is the minimal constraint (w.r.t. $\succeq$) entailing the same information (basic constraints) than e'.

We shall then show that for the monotonic fragment, the symbolic input-output relation is a closure operator [Scott 1982], i.e., an extensive, idempotent and monotonic function. A pleasant property of these functions is that they are uniquely determined by their set of fixed points. Then, we shall define the *strongest postcondition* of a process as the set of fixed points of the closure operator associated to its symbolic input-output relation.

Next, following the denotational semantics of `tcc` in [Saraswat 1994, Nielsen 2002a], we shall give a compositional characterization of the symbolic strongest postcondition relation. Recall that the symbolic outputs of a process are past-monotonic sequences (see Definition 4.2.3). Consequently, the codomain of our denotational model will be past-monotonic sequences unlike sequences of basic constraints as in `tcc`.

We shall prove our representation to be fully abstract with respect to the symbolic input-output behavior for a meaningful fragment of the calculus. This shows that mobility can be captured as closure operators over an underlying constraint system. Furthermore, we shall show that the input-output behavior of monotonic processes can be compositionally retrieved from its denotation.

As an application of the semantics here presented, in Chapter 8 we shall give a closure operator semantics to a language for the specification of security protocols that arises as a specialization of `utcc` with a cryptographic constraint system.

7.1 Symbolic Behavior as Closure Operators

As we studied in Chapter 4, the outputs of the symbolic and the operational semantics are rather different. On the one hand, operationally, only basic constraints can be output. This way, the environment observes exactly what the process computes in each time unit. On the other hand, the symbolic semantics outputs past-monotonic sequences which are an abstract representation of all the potential outputs the process can exhibit.

We have also shown that even if both representations are different, they coincide in the set of basic constraints they can entail when considering well-terminated processes. One may then wonder how to determine the actual behavior of a process by observing the

temporal formulae it outputs when providing an input. For example, one may expect that the process **when** c **do tell**(d) under input `true` outputs `true` (assuming $c \neq$ `true`) and not the implication $e = c \Rightarrow d$ (produced by the symbolic semantics). In fact, we know that only the basic constraint `true` can be entailed from e and then, the operational output `true` corresponds to the symbolic output e.

Fixed Formulae. Aiming at capturing such a "more concrete" representation of the output of a process, we shall define the symbolic input-output relation for `utcc` processes. This relation is based on the following idea. Assume that $P \xrightarrow{(d,e')}_s Q$, i.e., under input d, P produces symbolically e'. We shall define $Fix(e')$ as the set of formulae (constraints) such that e' cannot add any new information. This means, for any d' in $Fix(e')$, the formulae $e' \wedge d'$ and d' entail the same basic constraints. We then define the output of P under input d as the minimum element e of $Fix(e')$ greater than the input d. Later on, we prove that e and e' entail the same basic constraints, i.e., e and e' represent the same information.

Sequences of formulae and $\vec{x}$-variants. Before formalizing the ideas above, let us first introduce some notation and definitions about sequences of formulae. Recall that an infinite sequence of future-free formulae w is said to be *past-monotonic* if and only if for all $i > 1$, $w(i) \models_T \ominus w(i-1)$ (see Definition 4.2.3). Recall also that FF denotes the set of future-free formulae.

Notation 7.1.1. *We shall use $\exists_{\vec{x}} w$ to denote the sequence obtained by pointwise applying $\exists_{\vec{x}}$ to each constraint in w. Similarly, $w \wedge w'$ denotes the sequence v such that $v(i) = w(i) \wedge w'(i)$ for $i > 0$.*

Definition 7.1.1 ($\vec{x}$-variant)**.** *We say that e and e' are $\vec{x}$-variants if $\exists_{\vec{x}} e = \exists_{\vec{x}} e'$. Similarly, the sequence w is an $\vec{x}$-variant of the sequence w' iff $\exists_{\vec{x}} w = \exists_{\vec{x}} w'$.*

Recall that $adm(\vec{x}, \vec{t})$ means that none of the elements in $\vec{x}$ is syntactically equal to the elements in $\vec{t}$ (Convention 3.1.1). Lemma 7.1.1 characterizes a special form of $\vec{x}$-variants that we shall consider in the forthcoming proofs.

Lemma 7.1.1. *Let w be a sequence of future-free formulae s.t. $\vec{x} \notin fv(w)$ and $\vec{t} \in \mathcal{T}^{|\vec{x}|}$ be a sequence of terms s.t. $adm(\vec{x}, \vec{t})$. If w' is an $\vec{x}$-variant of w and $w' \succeq (\vec{x} = \vec{t})^\omega$ then $w' \equiv (w \wedge (\vec{x} = \vec{t})^\omega)$. Furthermore, if w, w' are past-monotonic sequences, then $w' \equiv (w \wedge \overline{(\vec{x} = \vec{t})^\omega})$.*

Proof. Let w' be an $\vec{x}$-variant of w such that $w' \succeq (\vec{x} = \vec{t})^\omega$. We can rewrite w' as

$$w' = w'' \wedge (\vec{x} = \vec{t})^\omega \wedge w'''$$

for some w'' and w''' s.t. $\vec{x} \notin fv(w'')$. We can substitute in w''' all the occurrences of $x_i \in \vec{x}$ by its corresponding $t_i \in \vec{t}$ obtaining $w' = v' \wedge (\vec{x} = \vec{t})^\omega$, where $v' = w'' \wedge w'''[\vec{t}/\vec{x}]$. From $\vec{x} \notin fv(w) \cup fv(v')$ and the definition of $\vec{x}$-variant we derive

$$w = \exists_{\vec{x}} w' = \exists_{\vec{x}}(v' \wedge (\vec{x} = \vec{t})^\omega) = \exists_{\vec{x}}(v') = v'$$

Then we conclude $w' \equiv (w \wedge (\vec{x} = \vec{t})^\omega)$.

Similarly we can prove the case when w and w' are past-monotonic sequences. □

Definition 7.1.2 (Fixed Formulae). *Let $n \geq 0$ and $Fix : FF \rightharpoonup \mathcal{P}(FF)$ be defined as*

$$\begin{array}{lcl} Fix(c) & = & \{F \in FF \mid F \models_T c\} \\ Fix(F_1 \wedge F_2) & = & \{F \in FF \mid F \in Fix(F_1) \text{ and } F \in Fix(F_2)\} \\ Fix(\forall_{\vec{x}} \ominus^n (c) \Rightarrow F_1) & = & \{F \in FF \mid \text{ for all } \vec{x}\text{-variant } F' \text{ of } F, \text{ if } F' \models_T (\ominus^n(c) \wedge \vec{x} = \vec{t}) \\ & & \quad \text{for some } \vec{t} \in \mathcal{T} \text{ s.t. } adm(\vec{x}, \vec{t}) \text{ then } F' \in Fix(F_1) \ \} \\ Fix(\exists_{\vec{x}} F_1) & = & \{F \in FF \mid \text{ there exists an } \vec{x}\text{-variant } F' \text{ of } F \text{ s.t. } F' \in Fix(F_1)\} \\ Fix(\ominus F_1) & = & \{F \in FF \mid F = \ominus F' \text{ and } F' \in Fix(F_1)\} \end{array}$$

Given the future-free formulae F and G, if $F \in Fix(G)$ we say that F is a fixed formula for G.

Roughly speaking, $F \in Fix(G)$ if the formula $F \wedge G$ entails the same basic constraints than F. It intuitively means that G cannot add new information to F (we shall formally prove this in Lemma 7.1.2).

Let us give some intuitions about the equations in Definition 7.1.2. If $F \models_T c$ then c cannot add new information to the formula F. F is a fixed formula for the conjunction $F_1 \wedge F_2$ if F is a fixed formula for both F_1 and F_2.

A formula F is a fixed formula for the implication $F_1 \Rightarrow F_2$ if either F does not entail the antecedent F_1 or F is a fixed formula for the consequence F_2. We extend this idea to universally quantified implications of the form $\forall_{\vec{x}}(F_1 \Rightarrow F_2)$ taking into account the substitutions making valid F_1. The intuition is that for any $\vec{x}$-variant F' of F, if $F' \models_T \ominus^n c \wedge \vec{x} = \vec{t}$, i.e., $F' \models_T F_1\sigma$ for an admissible substitution $\sigma = [\vec{t}/\vec{x}]$, then F' must be also a fixed formula for $F_2\sigma$.

Let F' be an $\vec{x}$-variant of F (i.e., $\exists_{\vec{x}} F = \exists_{\vec{x}} F'$). For the existentially quantified formula $G = \exists_{\vec{x}} F_1$, if F' cannot add any new information to F_1 then F cannot add any new information to G. Hence, F is a fixed formula for G if there exists an $\vec{x}$-variant F' of F such that F' is a fixed formula for F_1.

Finally, the formula $F = \ominus F'$ cannot add any information to $G = \ominus F_1$ if F' cannot add any new information to F_1.

Remark 7.1.1. *Notice that we defined $Fix(\cdot)$ explicitly for the subset of future-free formulae generated by the symbolic semantics that corresponds to the following syntax*

$$F, G, \ldots := c \mid F \wedge G \mid \forall_{\vec{x}}(\ominus^n(c) \Rightarrow F) \mid \exists_{\vec{x}} F \mid \ominus F.$$

where c is a basic constraint in the underlying constraint system and $n \geq 0$.

Let us now lift the definition of $Fix(\cdot)$ to sequences of future-free formulae.

Notation 7.1.2. *Let w and v be sequences of future-free formulae. We shall write $w \in Fix(v)$ whenever $w(i) \in Fix(v(i))$ for $i > 0$ and we say that w is a fixed sequence for v.*

7.1.1 Symbolic Input-Output Relation

Using the previous definition of fixed formulae, we define here the symbolic input-output relation of a process. Later on, we shall show that for the monotonic fragment, this relation is a closure operator [Scott 1982], i.e., a monotonic, extensive and idempotent function.

We shall use the following notation.

Notation 7.1.3 (Upper Closure). *The upper closure of a future-free formula e is the set $\{e' \mid e' \succeq e\}$ and we write $\uparrow e$. We extend this notion to sequences of future-free formulae by decreeing that $\uparrow w = \{w' \mid w' \succeq w\}$.*

The following definition makes precise our idea of the symbolic input-output behavior of a process. Recall that we use $\overline{w}$ to denote the past-monotonic sequence obtained from w by adding the necessary past information (see Notation 4.2.2).

Definition 7.1.3 (Symbolic Input-Output Relation)**.** *Let min be the minimum function wrt the order induced by $\succeq$. Given an abstracted-unless free process P, we define the symbolic-input output behavior of P as the set*

$$io_s(P) = \{(w, v) \mid P \stackrel{(w,v')}{\Longrightarrow} \text{ and } v = min(\uparrow \overline{w} \cap Fix(v'))\}$$

We say that P and Q are symbolically input-output equivalent, notation $P \approx_s^{io} Q$ iff $io_s(P) = io_s(P)$.

Closure Properties of $io_s(\cdot)$. We can show that for the monotonic fragment of `utcc`, the relation $io_s(\cdot)$ is a closure operator. In the following proposition we prove this.

Proposition 7.1.1 (Closure Properties)**.** *Let P be a monotonic `utcc` process. We have the following:*

(1) $io_s(P)$ is a function.

(2) $io_s(P)$ is a closure operator, namely it satisfies:

- **Extensiveness***: If $(w, v) \in io_s(P)$ then $v \succeq w$.*
- **Idempotence***: If $(w, v) \in io_s(P)$ then $(v, v) \in io_s(P)$.*
- **Monotonicity***: if $(w_1, v_1) \in io_s(P)$ and $w_2 \succeq w_1$, then there exists v_2 such that $(v_1, v_2) \in io_s(P)$ and $v_2 \succeq v_1$.*

Proof. The proof of (1) is immediate from Theorem 4.3.1. For (2), assume that $(w, v) \in io_s(P)$ and $P \stackrel{(w,w\wedge v')}{\Longrightarrow}$. By definition of $io_s(\cdot)$, it must be the case that

$$v = min(\uparrow \overline{w} \cap Fix(w \wedge v'))$$

Then we have the following.

- **Extensiveness**. It is easy to see that $\overline{w} \succeq w$. Since $v = min(\uparrow \overline{w} \cap Fix(w \wedge v'))$ we conclude $v \succeq w$.
- **Idempotence**. Assume that $(v, u) \in io_s(P)$. Then, we have $P \stackrel{(v,v\wedge u')}{\Longrightarrow}$ and $u = min(\uparrow \overline{v} \cap Fix(v \wedge u'))$. Since P is a monotonic process, by Lemma 4.3.6 we know that $u' = v'$. Then we have $u = min(\uparrow \overline{v} \cap Fix(v \wedge v'))$. By hypothesis we know that $v = min(\uparrow \overline{w} \cap Fix(w \wedge v'))$ and then $v \in Fix(v')$. Since v is a past-monotonic sequence, we also know that $\overline{v} = v$. We then conclude by noticing that it must be the case that $u = v$.
- **Monotonicity**. Let $w' \succeq w$ and assume that $(w', u') \in io_s(P)$. Then $P \stackrel{(w',w'\wedge u'')}{\Longrightarrow}$ and $u' = min(\uparrow \overline{w'} \cap Fix(w' \wedge u''))$. By appealing to Lemma 4.3.6, we know that $u'' = v'$ and then $u' = min(\uparrow \overline{w'} \cap Fix(w' \wedge v'))$. Since $v = min(\uparrow \overline{w} \cap Fix(w \wedge v'))$ and $w' \succeq w$ we conclude $u' \succeq v$.

□

A pleasant property of closure operators is that they are functions uniquely determined by their set of fixed points. We shall call that set of fixed points the *strongest postcondition* that we define as follows.

Definition 7.1.4 (Strongest Postcondition)**.** *Given a monotonic process P, the set $sp_s(P) = \{w \mid (w,w) \in io_s(P)\}$ denotes the* strongest postcondition *of P. Moreover, if $w \in sp_s(P)$, we say that w is a quiescent sequent for P, i.e. P under input w cannot add any information whatsoever. Define $P \sim_s^{sp} Q$ iff $sp_s(P) = sp_s(Q)$.*

The following proposition introduces a obvious fact on the strongest postcondition relation.

Proposition 7.1.2. *Given a monotonic process P and the past-monotonic sequences w, v we have if $(w,v) \in io_s(P)$ then $v \in sp_s(P)$.*

Proof. Directly from the idempotence of $io_s(\cdot)$. □

Finally, we give an alternative characterization of the strongest postcondition as the sequences w such that $P \xRightarrow{(w,v)}_s$ and w is a fixed sequence for v. The following proposition shows that this representation and that of Definition 7.1.4 are equivalent. Then, in the sequel, we shall use indistinguishably both definitions of the strongest postcondition.

Proposition 7.1.3 (Alternative Characterization of $sp_s(\cdot)$)**.** *Let P be a monotonic process.*

$$w \in sp_s(P) \text{ iff } P \xRightarrow{(w,v)}_s \text{ and } w \in Fix(v)$$

Proof. ($\Rightarrow$) Assume that $w \in sp_s(P)$. Then, w is a past-monotonic sequence and $\overline{w} = w$. Hence, it must be the case that $P \xRightarrow{(w,w\wedge v)}_s$ and $w = min(\uparrow w \cap Fix(w \wedge v))$. Then, we must have that $w \in Fix(w \wedge v)$. Hence, $w \in Fix(v)$.
($\Leftarrow$) Assume that $P \xRightarrow{(w,v)}_s$ and $w \in Fix(v)$. Then $w = min(\uparrow w \cap Fix(v))$ and we conclude $(w,w) \in io_s(P)$.

□

7.1.2 Retrieving the Input-Output Behavior

An interesting application of the fact that $io_s(P)$ is a closure operator is that this relation can be retrieved from its set of fixed points, i.e., from the relation $sp_s(P)$ similarly as in the case of `tcc` [Saraswat 1994, Nielsen 2002a].

Corollary 7.1.1 (Input-output Retrieval from sp_s)**.** *Given a monotonic `utcc` process P,*

$$(w,w') \in io_s(P) \text{ iff } w' = min(\uparrow \overline{w} \cap sp_s(P))$$

Proof. Directly from the fact that $io_s(\cdot)$ is a closure operator. □

Therefore, to characterize the input-output behavior of a monotonic process P, it suffices to specify $sp_s(P)$. In the next section we shall introduce a denotational semantics aiming at capturing $sp_s(\cdot)$ compositionally. Then, we can retrieve the symbolic input-output relation compositionally relaying on the previous corollary.

Properties of *Fix* Before presenting the denotational semantics for `utcc`, the following lemma proves our intuition that if $F \in Fix(G)$ then $F \wedge G$ and F entail the same basic constraints.

Lemma 7.1.2. *Let F, G be future free formulae and d be a basic constraint. If $F \in Fix(G)$, then $F \models_T d$ iff $F \wedge G \models_T d$.*

Proof. The *if* part is trivial. Concerning the *only-if* part we proceed by induction on the structure of G. We only present the cases for the existential and the universal quantification. The other cases are easier.

Assume that $F \in Fix(G)$ and $F \wedge G \models_T d$ where d is a basic constraint. In both cases below, by alpha conversion we shall assume that $\vec{x} \notin fv(F)$. Then, if $F \wedge G \models_T d$ we can also assume that $\vec{x} \notin fv(d)$.

- $G = \exists_{\vec{x}} F_1$. By definition of *Fix*, there exists an $\vec{x}$-variant F' of F s.t. $F' \in Fix(F_1)$. Since F and F' are $\vec{x}$-variants and $\vec{x} \notin fv(F)$, $F = \exists_{\vec{x}} F'$ and then $F' \models_T F$. We also have $F_1 \models_T G$ since $G = \exists_{\vec{x}} F_1$. Given that $F \wedge G \models_T d$ we derive the following

  $$F' \wedge F_1 \models_T F' \wedge G \models_T F \wedge G \models_T d$$

  By inductive hypothesis, if $F' \wedge F_1 \models_T d$ then $F' \models_T d$. From the assumption $\vec{x} \notin fv(d)$ we conclude $\exists_{\vec{x}} F' \models_T d$ and then $F \models_T d$.

- $G = \forall_{\vec{x}}(\ominus^n(c) \Rightarrow F_1)$ with $n \geq 0$. We shall prove for any model σ if $\sigma \models_T F \wedge G \Rightarrow d$ then it must be the case that $\sigma \models_T F \Rightarrow d$. We have to consider two cases: $\sigma \models_T d$ and $\sigma \not\models_T F \wedge G$:

  - If $\sigma \models_T d$ then trivially $\sigma \models_T F \Rightarrow d$.
  - If $\sigma \not\models_T F \wedge G$ then either $\sigma \not\models_T F$ or $\sigma \not\models_T G$. In the first case, trivially we have $\sigma \models_T F \Rightarrow d$. In the second case, there exists σ' $\vec{x}$-variant of σ such that $\sigma' \models_T \ominus^n(c)$ and $\sigma' \not\models_T F_1$. Then, we must have that $\sigma' \models_T F \wedge F_1 \Rightarrow d$. By inductive hypothesis, $\sigma' \models_T F \Rightarrow d$. Since $\vec{x} \notin fv(F) \cup fv(d)$ we conclude $\sigma \models_T F \Rightarrow d$.

□

Finally, we can prove that the set of basic constraints entailed from v in $(u, v) \in io_s(P)$ and v' in $P \xrightarrow{(w,v')}_s$ coincide.

Theorem 7.1.1 (Basic Constraints and Symbolic Outputs)**.** *Let P be a abstract-unless free process such that $P \xrightarrow{(w,v')}_s$ and $(w, v) \in io_s(P)$. For all $i > 0$ and basic constraint d, $v'(i) \models_T d$ iff $v(i) \models_T d$.*

Assume that $P \xrightarrow{(w,v')}_s$ and $(w, v) \in io_s(P)$. Then, it must be the case that $w \in Fix(v)$. Let $i > 0$ and $d_i = w(i)$, $e'_i = v'(i)$ and $e_i = v(i)$.

Proof. ($\Rightarrow$) Assume that $e'_i \models_T d$ and then $e'_i \wedge e_i \models_T d$. Since $e_i \in Fix(e'_i)$, by Lemma 7.1.2 we know that $e_i \models_T d$.

($\Leftarrow$) Assume that $e_i \models_T d$. By definition of $io_s(\cdot)$ we know that $v = min(\uparrow \overline{w} \cap Fix(v'))$. By extensiveness, it must be the case that $v' \succeq \overline{w}$. One can show that for all sequence v', $v' \in Fix(v')$. Therefore, $v' \succeq v$ and we conclude $e'_i \models_T d$.

□

7.2 Denotational Semantics for utcc

In this section we define a compositional semantics capturing the symbolic strongest postcondition in Definition 7.1.4. The semantics is defined as a function $[\![\cdot]\!]$ that associates to each process a set of sequences of past-monotonic formulae, namely $[\![\cdot]\!] : \mathit{Proc} \to \mathcal{P}(PM)$ (see Table 7.1).

Let us give some intuitions about the semantic equations. Recall that the strongest-postcondition of a process P is the set of sequences on input of which P can run without adding any information whatsoever. Since **skip** cannot add any information to any sequence in PM, then any past-monotonic sequence is quiescent for **skip** (Equation $\mathrm{D}_{\mathrm{SKIP}}$).

The sequences to which **tell**(c) cannot add information are those whose first element can entail c (Equation $\mathrm{D}_{\mathrm{TELL}}$).

The Equation $\mathrm{D}_{\mathrm{PAR}}$ says that a sequence is quiescent for $P \parallel Q$ if it is for P and Q.

A process **next** P has not influence on the first element of a sequence, thus $e.w$ is quiescent for it if w is quiescent for P (Equation $\mathrm{D}_{\mathrm{NEXT}}$). Similarly, a sequence $e.w$ is quiescent for **unless** c **next** P if either e entails c or w is quiescent for P (Equation $\mathrm{D}_{\mathrm{UNL}}$).

A sequence w is quiescent for $!\,P$ if w is quiescent for all process of the form **next**$^n P$ with $n \geq 0$. This implies that every suffix of w is quiescent for P (Equation $\mathrm{D}_{\mathrm{REP}}$).

Binding Processes. We now consider the binding processes. Recall that a sequence w is $\vec{x}$-variant of w' if $\exists_{\vec{x}} w = \exists_{\vec{x}} w'$ (Definition 7.1.1). A sequence w is quiescent for $Q = (\mathbf{local}\,\vec{x}; c)\,P$ if there exists an $\vec{x}$-variant w' of w such that w' is quiescent for P. Hence, if P cannot add any information to w' then Q cannot add any information to w. To see this, assume that w and w' are $\vec{x}$-variants. Clearly Q cannot add any information on (the global variables) $\vec{x}$ appearing in w. So, if Q were to add information to w, then P could also do the same to w'. But the latter is not possible since w' is quiescent for P.

Now, we may then expect that the semantics for the abstraction operator can be straightforwardly obtained in a similar way by quantifying over all possible $\vec{x}$-variants. Nevertheless, this is not the case as we shall show in the next section.

7.2.1 Semantic Equation for Abstractions Using $\vec{x}$-variants

Recall that the *ask* tcc process **when** c **do** Q is a shorthand for the empty abstraction process $(\mathbf{abs}\ \varepsilon; c)\,Q$ (Notation 3.3.1). Recall also that $\mathcal{T}$ denotes the set of all terms in the underlying constraint system. The first intuition for the denotation of the process $P = (\mathbf{abs}\ \vec{x}; c)\,Q$ arises directly from the fact that P can be viewed as the (possibly infinite) parallel composition of the processes $(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]$ for every sequence of terms $\vec{t} \in \mathcal{T}^{|\vec{x}|}$. Then we can give the following semantic equation for this operator:

$$[\![(\mathbf{abs}\ \vec{x}; c)\,P]\!] = \bigcap_{\vec{t} \in \mathcal{T}^{|\vec{x}|}} [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!] \tag{7.1}$$

where $[\![\mathbf{when}\ c\ \mathbf{do}\ Q]\!]$ is the denotational equation for the tcc *ask* operator [Saraswat 1994]:

$$[\![\mathbf{when}\ c\ \mathbf{do}\ Q]\!] = \{w \mid w(1) \models_T c \text{ implies } w \in [\![Q]\!]\}$$

Nevertheless, we can give a denotational equation for the abstraction operator which is analogous to that of the local operator. By using the notion of $\vec{x}$-variants, the equation does not appeal to substitutions as the one above. As illustrated in the example below, the denotation of the **abs** operator is not entirely dual to the denotation of the local operator. The lack of duality between $\mathrm{D}_{\mathrm{LOC}}$ and $\mathrm{D}_{\mathrm{ABS}}$ is reminiscent of the result in CCP

[de Boer 1997] stating that negation does not correspond exactly to the complementation (see [Cortesi 1997, de Boer 1997]).

Example 7.2.1. *Let $\overline{w}$ be as in Notation 4.2.2 and $Q = (\mathbf{abs}\ x; c)\, P$ where $c = \mathtt{out}\,(x)$ and $P = \mathbf{tell}(\mathtt{out}'(x))$. Take the past-monotonic sequence*

$$w = \overline{(\mathtt{out}(0) \wedge \mathtt{out}'(0)).\,\mathtt{true}^{\omega}} \in sp_s\,(Q).$$

Suppose that we were to define:

$$[\![Q]\!] = \{w \mid \textit{for every } x\textit{-variant } w' \textit{ of } w \textit{ if } w'(1) \models_T c \textit{ then } w' \in [\![P]\!]\} \tag{7.2}$$

Let $c' = \mathtt{out}(0) \wedge \mathtt{out}'(0) \wedge \mathtt{out}(x)$ and $w' = \overline{c'.\,\mathtt{true}^{\omega}}$. Notice that the w' is an x-variant of w, $w'(1) \models_T c$ but $w' \notin [\![P]\!]$ (since $c' \not\models_T \mathtt{out}'(x)$). Then $w \notin [\![Q]\!]$ under this naive definition of $[\![Q]\!]$.

We fix the Equation 7.2 by adding the extra condition that $w' \succeq (\vec{x} = \vec{t})^{\omega}$ for a sequence of terms $\vec{t}$ such that $|\vec{t}| = |\vec{x}|$ and $adm(\vec{x}, \vec{t})$ as in Equation $\mathrm{D}_{\mathrm{ABS}}$ (Table 7.1). Intuitively, this condition together with $w'(1) \models_T c$ requires that $w'(1) \models_T c \wedge \vec{x} = \vec{t}$ and hence that $w'(1) \models_T c\sigma$ for an admissible substitution $\sigma = [\vec{t}/\vec{x}]$. Furthermore $w' \succeq (\vec{x} = \vec{t})^{\omega}$ together with $w' \in [\![P]\!]$ realizes the operational intuition that P runs under the substitution σ.

We can prove the Equations $\mathrm{D}_{\mathrm{ABS}}$ and 7.1 to be equivalents. Before that we require the following Lemma.

Lemma 7.2.1. *Let $[\![\cdot]\!]$ be as in Table 7.1 and w be a past-monotonic sequence such that $\vec{x} \notin fv(w)$. For any process P and admissible substitution $[\vec{t}/\vec{x}]$ the following holds*

$$w \in [\![P[\vec{t}/\vec{x}]]\!] \quad \textit{iff } w \in [\![(\mathbf{local}\,\vec{x})\,(P \parallel\!!\, \mathbf{tell}(\vec{x} = \vec{t}))]\!]$$

Proof. ($\Rightarrow$). Assume $w \in [\![P[\vec{t}/\vec{x}]]\!]$ and $\vec{x} \notin fv(w)$ by alpha conversion. Since $\vec{x} \notin fv(P[\vec{t}/\vec{x}])$, we can prove that $w' = w \wedge \overline{(\vec{x} = \vec{t})^{\omega}} \in [\![P[\vec{t}/\vec{x}]]\!]$. Given that $\vec{x} \notin fv(w)$ then w' is an $\vec{x}$-variant of w. By Equation $\mathrm{D}_{\mathrm{TELL}}$, $w' \in [\![!\,\mathbf{tell}(\vec{x} = \vec{t})]\!]$ and by Equations $\mathrm{D}_{\mathrm{LOC}}$ and $\mathrm{D}_{\mathrm{PAR}}$, $w \in [\![(\mathbf{local}\,x)\,(P[\vec{t}/\vec{x}] \parallel\!!\, \mathbf{tell}(\vec{x} = \vec{t}))]\!]$. One can easily prove that $[\![P[\vec{t}/\vec{x}] \parallel\!!\, \mathbf{tell}(\vec{x} = \vec{t})]\!] = [\![P \parallel\!!\, \mathbf{tell}(\vec{x} = \vec{t})]\!]$. Therefore, $w \in [\![(\mathbf{local}\,x)\,(P \parallel\!!\, \mathbf{tell}(\vec{x} = \vec{t}))]\!]$.

The proof of ($\Leftarrow$) can be obtained easily by reversing the previous steps. □

The following proposition shows the Equations $\mathrm{R}_{\mathrm{ABS}}$ and 7.1 to be equivalents.

Proposition 7.2.1. *Let $[\![\cdot]\!]$ be as in Table 7.1 and $P = (\mathbf{abs}\ \vec{x}; c)\, Q$.*

$$w \in [\![P]\!] \quad \textit{iff } w \in \bigcap_{\vec{t} \in \mathcal{T}^{|\vec{x}|}} [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]$$

Proof. By alpha-conversion we can assume $\vec{x} \notin fv(w)$.

($\Rightarrow$) As a mean of contradiction, assume that $w \in [\![P]\!]$ and

$$w \notin \bigcap_{\vec{t} \in \mathcal{T}^{|\vec{x}|}} [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]$$

Then by definition of $[\![\mathbf{when}\ c\ \mathbf{do}\ Q]\!]$ there exists $\vec{t'} \in \mathcal{T}^{|\vec{x}|}$ such that $w(1) \models_T c[\vec{t'}/\vec{x}]$ and $w \notin [\![Q[\vec{t'}/\vec{x}]]\!]$. By Lemma 7.2.1, $w \notin [\![(\mathbf{local}\,\vec{x})\,(Q \parallel\!!\, \mathbf{tell}(\vec{x} = \vec{t'}))]\!]$ and then, there is not an $\vec{x}$-variant w' of w such that $w' \in [\![Q]\!] \cap [\![!\,\mathbf{tell}(\vec{x} = \vec{t'})]\!]$ (Rules $\mathrm{D}_{\mathrm{LOC}}$ and $\mathrm{D}_{\mathrm{PAR}}$). We have to consider two cases:

$\mathrm{D}_{\mathrm{SKIP}}$	$[\![\mathbf{skip}]\!]$	$=$	PM
$\mathrm{D}_{\mathrm{TELL}}$	$[\![\mathbf{tell}(c)]\!]$	$=$	$\{e.w \in PM \mid e \models_T c\}$
$\mathrm{D}_{\mathrm{PAR}}$	$[\![P \parallel Q]\!]$	$=$	$[\![P]\!] \cap [\![Q]\!]$
$\mathrm{D}_{\mathrm{NEXT}}$	$[\![\mathbf{next}\, P]\!]$	$=$	$\{e.w \in PM \mid w \in [\![P]\!]\}$
$\mathrm{D}_{\mathrm{UNL}}$	$[\![\mathbf{unless}\ c\ \mathbf{next}\, P]\!]$	$=$	$\{e.w \in PM \mid e \not\models_T c \text{ and } w \in [\![P]\!]\} \cup$ $\{e.w \in PM \mid e \models_T c\}$
$\mathrm{D}_{\mathrm{REP}}$	$[\![!\, P]\!]$	$=$	$\{w \in PM \mid \text{for all } v, v' \text{ s.t. } w = v.v',\ v' \in [\![P]\!]\}$
$\mathrm{D}_{\mathrm{LOC}}$	$[\![(\mathbf{local}\, \vec{x}; c)\, P]\!]$	$=$	$\{w \in PM \mid \text{there exists an } \vec{x}\text{-variant}$ $w' \text{ of } w \text{ s.t } w'(1) \models_T c \text{ and } w' \in [\![P]\!]\}$
$\mathrm{D}_{\mathrm{ABS}}$	$[\![(\mathbf{abs}\ \vec{x}; c)\, P]\!]$	$=$	$\{w \in PM \mid \text{for every } \vec{x}\text{-variant } w' \text{ of } w \text{ if } w'(1) \models_T c$ $\text{and } w' \succeq (\vec{x} = \vec{t})^\omega \text{ for some } \vec{t} \text{ s.t. } \lvert\vec{x}\rvert = \lvert\vec{t}\rvert \text{ and}$ $adm(\vec{x}, \vec{t}) \text{ then } w' \in [\![P]\!]\}$

Table 7.1: Denotational Semantics for utcc. The function $[\![\cdot]\!]$ is of type $Proc \to \mathcal{P}(PM)$. In $\mathrm{D}_{\mathrm{ABS}}$, $\vec{x} = \vec{t}$ denotes the constraint $\bigwedge_{1 \le i \le |\vec{x}|} x_i = t_i$ and $adm(\vec{x}, \vec{t})$ is as in Convention 3.1.1. If $|\vec{x}| = 0$ then $\vec{x} = \vec{t}$ is defined as `true`.

- Assume $adm(\vec{x}, \vec{t'})$. Let $w'' = w[\vec{t'}/\vec{x}] \wedge \overline{(\vec{x} = \vec{t'})^\omega}$. Since $\vec{x} \notin fv(w)$, w'' is an $\vec{x}$-variant of w. From $w'' \succeq (\vec{x} = \vec{t'})^\omega$ and $w(1) \models_T c[\vec{t'}/\vec{x}]$ we have $w''(1) \models_T c[\vec{t'}/\vec{x}]$. By equation $\mathrm{D}_{\mathrm{ABS}}$, $w'' \in [\![Q]\!] \cap [\![!\, \mathbf{tell}(\vec{x} = \vec{t'})]\!]$ thus a contradiction.

- Assume that $t'_i \doteq x_j$ for some i, j. In this case the $\vec{x}$-variant $w'' = w[\vec{t'}/\vec{x}] \wedge \overline{(\vec{x} = \vec{t'})^\omega}$ as above does not satisfy the condition $w'' \not\succeq (\vec{x} = \vec{t})^\omega$ for a $\vec{t}$ admissible for $\vec{x}$. Then, we cannot use directly the equation $\mathrm{D}_{\mathrm{ABS}}$. Let $\vec{x} = x_1, ..., x_n$ and $\vec{y} = y_1, .., y_n$ such that y_i does not occur neither in w nor in P. Let P' be as P but renaming each x_i by y_i. Then $P \equiv P'$ (alpha-conversion) and $w \in [\![P']\!]$. Since $\vec{x} \cup \vec{y} \notin fv(w)$, we proceed as in the previous case by using the $\vec{x}$-variant (and $\vec{y}$-variant) $w'' = w[\vec{t'}/\vec{x}] \wedge \overline{(\vec{y} = \vec{t'})^\omega}$.

($\Leftarrow$) As a mean of contradiction, assume that $w \in \bigcap_{\vec{t} \in \mathcal{T}^{|\vec{x}|}} [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]$ and there exists a $\vec{t}$ admissible for $\vec{x}$ s.t. w' is an $\vec{x}$-variant of w, $w'(1) \models_T c$, $w' \succeq (\vec{x} = \vec{t'})^\omega$ and $w' \notin [\![Q]\!]$. By hypothesis and Lemma 7.2.1 we have

$$w \in [\![(\mathbf{local}\, \vec{x})\, (\mathbf{when}\ c\ \mathbf{do}\ Q \parallel !\, \mathbf{tell}(\vec{x} = \vec{t'}))]\!]$$

Hence, there exists an $\vec{x}$-variant w'' of w such that $w'' \in [\![\mathbf{when}\ c\ \mathbf{do}\ Q]\!] \cap [\![!\, \mathbf{tell}(\vec{x} = \vec{t'})]\!]$. Then, $w', w'' \succeq (\vec{x} = \vec{t'})^\omega$ and by Lemma 7.1.1 $w' = w'' = w \wedge \overline{(\vec{x} = \vec{t})^\omega}$. By hypothesis, $w'(1) \models_T c$, so $w''(1)$. Then, since $w'' \in [\![\mathbf{when}\ c\ \mathbf{do}\ Q]\!]$, $w', w'' \in [\![Q]\!]$, thus a contradiction. □

7.3 Full Abstraction

In this section we shall prove our denotational model to be fully abstract with respect to the symbolic input-output behavior for the locally-independent and abstracted-unless free fragment of the calculus (see Definition 7.3.1). We first prove the soundness of the denotation, i.e., $sp_s(P) \subseteq [\![P]\!]$, in Section 7.3.1. Later on, in Section 7.3.2, we prove completeness, i.e., $[\![P]\!] \subseteq sp_s(P)$. Finally, we show that the symbolic input-output behavior of a monotonic process (i.e. a process without occurrences of **unless** processes) can be compositionally retrieved from its denotation.

7.3.1 Soundness of the Denotation

As we shall see, the proof of the soundness theorem proceeds by induction on the structure of the process. For the cases of the abstraction and the local operator, we shall require some auxiliary results.

The following proposition relates the strongest postcondition of the process $P = (\mathbf{abs}\ \vec{x}; c)\, Q$ and Q.

Proposition 7.3.1. *Let $P = (\mathbf{abs}\ \vec{x}; c)\, Q$ be an abstracted-unless free process and w be a past-monotonic sequence. The following statements are equivalent*

1. $w \in sp_s(P)$
2. *For all w' $\vec{x}$-variant of w s.t. $w' \succeq (\vec{x} = \vec{t})^\omega$ for a $\vec{t}$ admissible for $\vec{x}$, if $w' \models_T c$ then $w' \in sp_s(Q)$.*

Proof. Assume by alpha-conversion that $\vec{x} \notin fv(w)$ and assume the following derivation:

$$P = P_1 \overset{(e_1, e_1 \wedge d_1)}{\Longrightarrow}_s P_2 \overset{(e_2, e_\wedge d_2)}{\Longrightarrow}_s P_3 \overset{(e_3, e_3 \wedge d_3)}{\Longrightarrow}_s \dots$$

Let $w = e_1.e_2.e_3...$, $v = d_1.d_2.d_3...$ and $v' = d'_1.d'_2.d'_3....$ Assume the following derivation:

$$Q = Q_1 \overset{(e_1, e_1 \wedge d'_1)}{\Longrightarrow}_s Q_2 \overset{(e_2, e_2 \wedge d'_2)}{\Longrightarrow}_s Q_3 \overset{(e_3, e_3 \wedge d'_3)}{\Longrightarrow}_s \dots$$

By the rule $\mathrm{R}_{\mathrm{ABS-SYM}}$ and the fact that $\vec{x} \notin fv(w)$, we must have that $d_1 = \forall_{\vec{x}}(c \Rightarrow d'_1)$ and for all $i > 1$, $d_i = \forall_{\vec{x}}(\ominus^{i-1}(c) \Rightarrow d'_i) \wedge \ominus(d_{i-1})$. Furthermore, since Q is a monotonic process, by Lemma 4.3.8 we know that $Q \overset{(w', w' \wedge v')}{\Longrightarrow}_s$ for any w'.

($\Rightarrow$) Assume that $w \in sp_s(P)$ and then $w \in Fix(v)$. Let w' be an $\vec{x}$-variant of w s.t. $w' \models_T c$, $w' \succeq (\vec{x} = \vec{t})^\omega$ and $adm(\vec{x}, \vec{t})$. Since $w' \models_T c$ and $w \in Fix(v)$ we know that $w' \in Fix(v')$. We conclude that $w' \in sp_s(Q)$ by noticing that $Q \overset{(w', w' \wedge v')}{\Longrightarrow}_s$.

($\Leftarrow$) Since $\vec{x} \notin fv(w)$, by Lemma 7.1.1 we know that if w' is an $\vec{x}$-variant of w such that $w' \succeq (\vec{x} = \vec{t})^\omega$ and $adm(\vec{x}, \vec{t})$, it must be the case that $w' = w \wedge \overline{(\vec{x} = \vec{t})^\omega}$. Let $w' = w \wedge \overline{(\vec{x} = \vec{t})^\omega}$ for an arbitrary $\vec{t} \in \mathcal{T}^{|\vec{x}|}$ admissible for $\vec{x}$. If $w' \models_T c$ we know by hypothesis that $w' \in sp_s(Q)$ and then $w' \in Fix(v')$. Since for every w' satisfying the conditions above, $w' \in Fix(v')$ we conclude that $w \in Fix(v)$ and then $w \in sp_s(P)$.

□

Now we state an auxiliary result that we shall use for the proof of the case of the local operator. We shall prove that for a process of the form $Q = (\mathbf{local}\, \vec{x})\, P$ where P is a monotonic process, if $w \in sp_s(Q)$ then there exists w' $\vec{x}$-variant of w such that $w' \in sp_s(P)$.

Proposition 7.3.2. *Let $P = (\mathbf{local}\,\vec{x}; c)\,Q$ such that Q is a monotonic process and let w be a past-monotonic sequence such that $w \in sp_s(P)$. Then, there exists w' $\vec{x}$-variant of w such that $w' \in sp_s(Q)$.*

Proof. Assume by alpha-conversion that $\vec{x} \notin fv(w)$. Assume also that $P \xRightarrow{(w, w \wedge \exists_{\vec{x}} u)}_s$ and $w \in sp_s(P)$. Therefore, $w \in Fix(\exists_{\vec{x}} u)$. By definition of $Fix(\cdot)$, there exists w' $\vec{x}$-variant of w such that $w'(1) \models_T c$ and $w' \in Fix(u)$. Since $P \xRightarrow{(w, w \wedge \exists_{\vec{x}} u)}_s$, by using the rule $\mathrm{R}_{\mathrm{LOC}}$ one can show that $Q \xRightarrow{(v, v \wedge u)}$ where $v = w \wedge \overline{c.\,\mathtt{true}^{\omega}}$. Since Q is a monotonic process, by Lemma 4.3.5 we have $Q \xRightarrow{(w', w' \wedge u)}$ and we conclude that $w' \in sp_s(Q)$ by noticing that $w' \in Fix(u)$.

□

In the previous lemma, we assumed the process Q in $(\mathbf{local}\,\vec{x}; c)\,Q$ is monotonic. Henceforth, we shall call locally-independent any process that verifies such a property.

Definition 7.3.1 (Locally Independent Processes)**.** *We say that P is locally independent iff P has no occurrences of processes of the form* **unless** c **next** Q *under the scope of a local operator.*

The following proposition introduces an obvious fact on the locally independent fragment of the calculus.

Proposition 7.3.3 (Locally-Independence Invariance)**.** *Let P be a locally-independent process. If $P \xRightarrow{(e,d)}_s Q$ then Q is also locally-independent.*

Proof. By induction on the structure of P and the definition of the symbolic future function F_s. □

Now we are ready to state the soundness of the denotation.

Theorem 7.3.1 (Soundness)**.** *Given a locally-independent and abstracted-unless free process P, $sp_s(P) \subseteq [\![P]\!]$.*

Proof. Assume that $w \in sp_s(P)$. The proof proceeds by induction on the structure of P.

- $P = \mathbf{skip}$. This case is trivial.
- $P = \mathbf{tell}(c)$. Let $w = e.v$. We must have: $\mathbf{tell}(c) \xRightarrow{(e, e \wedge c)}_s Q \xRightarrow{(v, v')}_s$ for some Q and v'. Since $w \in sp_s(P)$ then $e \in Fix(c)$ and it must be the case that $e \models_T c$. Hence by definition of $[\![\mathbf{tell}(c)]\!]$ we conclude $w \in [\![P]\!]$.
- $P = Q \parallel R$. Let $w = e_1.e_2.e_3...$ and $Q = Q_1$ and $R = R_1$. Assume the following derivation

$$Q_1 \parallel R_1 \xRightarrow{(e_1, e_1 \wedge d_1 \wedge g_1)}_s Q_2 \parallel R_2 \xRightarrow{(e_2, e_2 \wedge d_2 \wedge g_2)}_s Q_3 \parallel R_3 \ldots$$

  Such that for $i > 0$, each Q_{i+1} (resp. R_{i+1}) is an evolution of Q_i (resp. R_i) and d_i (resp. g_i) is the output of Q_i (resp. R_i) . By hypothesis $w \in sp_s(Q \parallel R)$ and for $i > 0$, $e_i \in Fix(d_i)$ and $e_i \in Fix(g_i)$. By using Lemma 7.1.2 we know that for all basic constraint c, $e_i \wedge d_i \wedge g_i \models_T c$ iff $e_i \wedge d_i \models_T c$ iff $e_i \wedge g_i \models_T c$. By using Lemma 4.3.4, one can show that there exists $Q_1 = Q'_1, Q'_2, Q'_3, ...$ and $R_1 = R'_1, R'_2, R'_3, ...$ s.t.

$$Q'_1 \xRightarrow{(e_1, e_1 \wedge d_1)}_s Q'_2 \xRightarrow{(e_2, e_2 \wedge d_2)}_s Q'_3 \xRightarrow{(e_3, e_3 \wedge d_3)}_s \ldots$$

and

$$R'_1 \xrightarrow{(e_1, e_1 \wedge g_1)}_s R'_2 \xrightarrow{(e_2, e_2 \wedge g_2)}_s R'_3 \xrightarrow{(e_3, e_3 \wedge g_3)}_s \ldots$$

From the fact that for $i > 0$, $e_i \in Fix(d_i)$ and $e_i \in Fix(g_i)$ we conclude $w \in sp_s(Q)$ and $w \in sp_s(R)$. By inductive hypothesis, $w \in [\![Q]\!]$ and $w \in [\![R]\!]$ and then $w \in [\![P]\!]$.

- $P = (\mathbf{abs}\ \vec{x}; c)\, Q$. By using alpha-conversion we can assume $\vec{x} \notin fv(w)$. Let $w = e_1.e_2.e_3...$ and $v = d_1.d_2.d_3...$ and assume the following derivation of P

$$P = P_1 \xrightarrow{(e_1, e_1 \wedge d_1)}_s P_2 \xrightarrow{(e_2, e_2 \wedge d_2)}_s P_3 \xrightarrow{(e_3, e_3 \wedge d_3)}_s ...$$

By hypothesis $w \in sp_s(P)$ and then $w \in Fix(v)$. Let $v' = d'_1.d'_2.d'_3...$ and assume the following derivation of Q

$$Q = Q_1 \xrightarrow{(e_1, e_1 \wedge d'_1)}_s Q_2 \xrightarrow{(e_2, e_2 \wedge d'_2)}_s Q_3 \xrightarrow{(e_3, e_3 \wedge d'_3)}_s ...$$

where $d_1 = \forall_{\vec{x}}(c \Rightarrow d'_1)$ and for $i > 1$, $d_i = \ominus(d_{i-1}) \wedge \forall_{\vec{x}}(\ominus^{i-1}(c) \Rightarrow d'_i)$.

Let w' be an arbitrary $\vec{x}$-variant of w such that $w' \succeq (\vec{x} = \vec{t})^\omega$ for a $\vec{t}$ admissible for $\vec{x}$ and $w'(1) \models_T c$. Since Q is monotonic, by Lemma 4.3.8 we know that $Q \xrightarrow{(w', w' \wedge v')}_s$. By Proposition 7.3.1, it must be the case that $w' \in sp_s(Q)$ and then, $w' \in Fix(v')$. Therefore, by appealing to induction, $w' \in [\![Q]\!]$ and we conclude.

- $P = (\mathbf{local}\, \vec{x}; c)\, Q$. Since P is a locally-independent process then Q is monotonic. By Proposition 7.3.2 we know that there exists w' $\vec{x}$-variant of w such that $w'(1) \models_T c$ and $w' \in sp_s(Q)$. By induction we know that $w' \in [\![Q]\!]$ and then $w \in [\![(\mathbf{local}\, \vec{x}; c)\, Q]\!]$.

- $P = \mathbf{next}\, Q$. Let $w = e.w'$. Then

$$P \xrightarrow{(e,e)}_s Q \xrightarrow{(w', w'')}_s$$

By hypothesis, $w' \in Fix(w'')$ and then $w' \in sp_s(Q)$. By inductive hypothesis $w' \in [\![Q]\!]$ and by definition of $[\![\mathbf{next}\, Q]\!]$, $w \in [\![P]\!]$.

- $P = \mathbf{unless}\ c\ \mathbf{next}\, Q$. We distinguish two cases:

  1. $w(1) \models c$. Immediate
  2. $w(1) \not\models c$. This case is similar to the case of $P = \mathbf{next}\, Q$.

- $P = !\, Q$. Let $w = e_1.e_2.e_3...$, $w' = e'_1.e'_2.e'_3....$ We can verify that

$$\langle P, e_1 \rangle \longrightarrow_s \langle Q \parallel \mathbf{next}\, !\, Q \rangle \longrightarrow_s^* \langle Q' \parallel \mathbf{next}\, !\, Q, e'_1 \rangle$$

We then have the following derivation for $Q = Q_{1,1}$:

$$\begin{array}{lll}
!\, Q_{1,1} & \xrightarrow{(e_1, e'_1)}_s & Q_{1,2} \parallel !\, Q_{1,1} \\
& \xrightarrow{(e_2, e'_2)}_s & Q_{1,3} \parallel Q_{2,2} \parallel !\, Q_{1,1} \\
& \xrightarrow{(e_3, e'_3)}_s & Q_{1,4} \parallel Q_{2,3} \parallel Q_{3,2} \parallel !\, Q_{1,1} \\
& \ldots & \\
& \xrightarrow{(e_{n-1}, e'_{n-1})}_s & Q_{1,n} \parallel Q_{2,n-1} \parallel Q_{3,n-2} \parallel \cdots \parallel Q_{n-1,2} !\, Q_{1,1} \\
& \ldots &
\end{array}$$

where each parallel component contributes in the following way:

$$
\begin{array}{l}
Q_{1,1} \xrightarrow{(e_1,e'_1)}_s Q_{1,2} \xrightarrow{(e_2,e'_2)}_s Q_{1,3} \cdots \xrightarrow{(e_{n-1},e'_{n-1})}_s Q_{1,n} \xrightarrow{(e_n,e'_n)}_s \cdots \\
Q_{1,1} \xrightarrow{(e_2,e'_2)}_s Q_{2,2} \xrightarrow{(e_3,e'_3)}_s Q_{2,3} \cdots \xrightarrow{(e_{n-1},e'_{n-1})}_s Q_{2,n-1} \xrightarrow{(e_n,e'_n)}_s \cdots \\
Q_{1,1} \xrightarrow{(e_3,e'_3)}_s Q_{3,2} \xrightarrow{(e_4,e'_4)}_s Q_{3,3} \cdots \xrightarrow{(e_{n-1},e'_{n-1})}_s Q_{3,n-2} \xrightarrow{(e_n,e'_n)}_s \cdots \\
\cdots
\end{array}
$$

By hypotheses, $w \in Fix(w')$ and then

$$
\begin{array}{r}
e_1.e_2.e_3...e_n... \in [\![Q]\!] \\
e_2.e_3...e_n... \in [\![Q]\!] \\
e_3...e_n... \in [\![Q]\!] \\
\cdots
\end{array}
$$

By equation $\mathrm{D_{REP}}$ we conclude $w \in [\![Q]\!]$

□

Remark 7.3.1. *In* tcc*, the locally-independent condition is required only for completeness and not for soundness. For the proof of the case* $P = (\mathbf{local}\,\vec{x}; c)\,Q$ *above, we appealed to Proposition 7.3.2. Therefore, we required* Q *to be a monotonic process. The technical problem is that from* $w \in Fix(\exists_{\vec{x}} w')$ *we can only deduce that there exists* w'' $\vec{x}$*-variant of* w *such that* $w'' \in Fix(w')$*. Nevertheless, this is not enough to prove that* $w'' \in sp_s(Q)$*. We believe that we can dispense with this restriction by finding an alternative way to prove the Proposition 7.3.2 for an arbitrary* Q*.*

7.3.2 Completeness of the Denotation

In this section we prove the completeness of the denotation, i.e., $[\![P]\!] \subseteq sp_s(P)$. For this result we have similar technical problems that in the case of tcc, namely: the combination between the **local** and the **unless** operator (see [Nielsen 2002a] for details).

Take for example $P = \mathbf{unless}\ x = a\ \mathbf{next}\ \mathbf{tell}(d)$ and $Q = (\mathbf{local}\,x)\,P$. Under input $w = (x = a).\,\mathtt{true}^\omega$, P outputs $w' = \overline{w}$ and under input $\mathtt{true}^\omega$ it outputs $w'' = \overline{\mathtt{true}.d.\,\mathtt{true}^\omega}$. Then $w' \in [\![Q]\!]$ but $w' \notin sp_s(Q)$. Thus, we shall prove the completeness of the denotation for the locally-independent fragment of utcc.

Theorem 7.3.2 (Completeness)**.** *Given a locally independent and abstracted-unless free process* P*,* $[\![P]\!] \subseteq sp_s(P)$

Proof. Assume that $w \in [\![P]\!]$. We proceed by induction on the structure of P:

- **skip**. This case is trivial
- $P = \mathbf{tell}(c)$. Let $w = e.w'$ with $e \models_T c$. Hence we have

$$P \xrightarrow{(e,e)}_s \mathbf{tell}(\ominus e) \xrightarrow{(w',w')}_s$$

Since w is a past-monotonic sequence, $w'(i) \models_T \ominus^i(e)$ for $i > 0$. Then we conclude $w \in sp_s(P)$

- $P = Q \parallel R$. Since $w \in [\![P]\!]$ we know that $w \in [\![R]\!]$, $w \in [\![Q]\!]$ and by hypothesis, $w \in sp_s(R)$ and $w \in sp_s(Q)$. Assume the following derivations of Q and R:

$$\begin{array}{l} Q = Q_1 \xrightarrow{(e_1, e_1 \wedge d_1)}_s Q_2 \xrightarrow{(e_2, e_2 \wedge d_2)}_s Q_3 \xrightarrow{(e_3, e_3 \wedge d_3)}_s \ldots \\ R = R_1 \xrightarrow{(e_1, e_1 \wedge g_1)}_s R_2 \xrightarrow{(e_2, e_2 \wedge g_2)}_s R_3 \xrightarrow{(e_3, e_3 \wedge g_3)}_s \ldots \end{array}$$

  Let $v = e_1.e_2.e_3...$, $v = d_1.d_2.d_3...$ and $u = g_1.g_2.g_3....$ Since $w \in sp_s(R)$ and $w \in sp_s(Q)$, we know that $w \in Fix(v)$ and $w \in Fix(u)$ and then $w \in Fix(v \wedge u)$. By Lemma 7.1.2, for $i > 0$ if $e_i \wedge d_i \wedge g_i \models_T c$ for a basic constraint c, it must be the case that $e_i \models_T c$. Then, by appealing to Lemma 4.3.4 one can prove that there exists $Q = Q'_1, Q'_2, Q'_3, ...$ and $R = R'_1, R'_2, R'_3, ...$ such that

$$Q'_1 \parallel R'_1 \xrightarrow{(e_1, e_1 \wedge d_1 \wedge g_1)}_s Q'_2 \parallel R'_2 \xrightarrow{(e_2, e_2 \wedge d_2 \wedge g_2)}_s Q'_3 \parallel R'_3 \xrightarrow{(e_3, e_3 \wedge d_3 \wedge g_3)}_s \ldots$$

  From $w \in Fix(v \wedge u)$ we conclude $w \in sp_s(Q \parallel R)$.

- $P = (\mathbf{abs}\ \vec{x}; c)\, Q$. By alpha-conversion we can assume $\vec{x} \notin fv(w)$. Let w' be an arbitrary $\vec{x}$-variant of w such that $w' \succeq (\vec{x} = \vec{t})^\omega$, $w'(1) \models_T c$ and $adm(\vec{x}, \vec{t})$. From the assumption $w \in [\![P]\!]$ we know that $w' \in [\![Q]\!]$. Appealing to induction we deduce that $w' \in sp_s(Q)$. Since for every $\vec{x}$-variant w' of w satisfying the conditions above we have $w' \in sp_s(Q)$, we can use Proposition 7.3.1 to conclude $w \in sp_s(P)$.

- $P = (\mathbf{local}\, \vec{x}; c)\, Q$. Let $w = e_1.e_2.e_3...$ and by alpha conversion assume that $\vec{x} \notin fv(w)$. From the assumption $w \in [\![P]\!]$, it must be the case that there exists $w' = e'_1.e'_2.e'_3...$ $\vec{x}$-variant of w such that $e'_1 \models_T c$ and $w' \in [\![Q]\!]$. By induction we know that $w' \in sp_s(Q)$. Assume the following derivation

$$Q = Q_1 \xrightarrow{(e'_1, e'_1 \wedge d'_1)}_s Q_2 \xrightarrow{(e'_2, e'_2 \wedge d'_2)}_s Q_3 \xrightarrow{(e'_3, e'_3 \wedge d'_3)}_s \ldots$$

  Let $v' = d'_1.d'_2.d'_3....$ Since P is a locally-independent process then Q is monotonic. Let $v'' = \overline{c.\mathtt{true}^\omega} = d''_1.d''_2.d''_3....$ By Lemma 4.3.6 we can show that there exists $Q = Q'_1, Q'_2, Q'_3, ...$ such that

$$Q'_1 \xrightarrow{(e_1 \wedge d''_1, e_1 \wedge d''_1 \wedge d'_1)}_s Q'_2 \xrightarrow{(e_2 \wedge d''_2, e_2 \wedge d''_2 \wedge d'_2)}_s Q'_3 \xrightarrow{(e_3 \wedge d''_3, e_3 \wedge d''_3 \wedge d'_3)}_s \ldots$$

  Given that $\vec{x} \notin fv(w)$, by using the rule $\mathrm{R_{LOC}}$ we can show that

$$P = P_1 \xrightarrow{(e_1, e_1 \wedge \exists_{\vec{x}}(d'_1 \wedge d''_1))}_s Q_2 \xrightarrow{(e_2, e_2 \wedge \exists_{\vec{x}}(d'_2 \wedge d''_2))}_s Q_3 \xrightarrow{(e_3, e_3 \wedge \exists_{\vec{x}}(d'_3 \wedge d''_3))}_s \ldots$$

  Since $w' \in sp_s(Q)$ then $w' \in Fix(v')$. Furthermore, given that $w'(1) \models_T c$ then $w' \in Fix(v'')$ and thus $w' \in Fix(v' \wedge v'')$. Since w' is an $\vec{x}$-variant of w we derive $w \in Fix(\exists_{\vec{x}}(v' \wedge v''))$ and then, $w \in sp_s(P)$.

- $P = \mathbf{next}\, Q$. Let $w = e.w'$. We have $P \xrightarrow{(e,e)}_s Q \parallel \mathbf{tell}(\ominus(e))$. Since w is a past-monotonic sequence, then $w'(1) \models_T \ominus(e)$. We know that $w' \in [\![Q]\!]$ and by induction $w' \in sp_s(Q)$. Therefore, if $\mathbf{next}\, Q \xrightarrow{(w,v')}_s$ it must be the case that $w \in Fix(v')$. Therefore $w \in sp_s(P)$.

- $P = \mathbf{unless}\ c\ \mathbf{next}\, Q$. Let $w = e.w'$. We distinguish two cases:

  1. $e \models c$. Then $P \xrightarrow{(e,e)}_s \mathbf{skip}$. Since $\mathbf{skip} \xrightarrow{(w',w')}_s$ then $w \in sp_s(P)$

2. $e \not\models c$ and $w' \in [\![Q]\!]$. We have $P \xrightarrow{(e,e)}_s Q$ and by inductive hypothesis, $w' \in sp_s(Q)$. Then we conclude $w \in sp_s(P)$

- $P =! Q$. Let $w = e_1.e_2.e_3....$ By definition of $[\![! Q]\!]$ we have

$$\begin{aligned} e_1.e_2.e_3...e_n... &\in [\![Q]\!] \\ e_2.e_3...e_n... &\in [\![Q]\!] \\ e_3...e_n... &\in [\![Q]\!] \\ &\dots \end{aligned}$$

By inductive hypothesis we have the following:

$$\begin{aligned} e_1.e_2.e_3...e_n... &\in sp_s(Q) \\ e_2.e_3...e_n... &\in sp_s(Q) \\ e_3...e_n... &\in sp_s(Q) \\ &\dots \end{aligned}$$

By a reasoning similar to the case of $P =! Q$ in Theorem 7.3.1, we conclude $w \in sp_s(P)$

□

From the soundness and the completeness results and the symbolic input-output characterization of `utcc` by means of its strongest postcondition in Corollary 7.1.1, we conclude by stating the full abstraction of our semantics.

Theorem 7.3.3 (Full abstraction)**.** *Let P and Q be a monotonic processes and $[\![\cdot]\!]$ as in Table 7.1. Then*

$$P \approx_s^{io} Q \quad \textit{iff } [\![P]\!] = [\![Q]\!].$$

Proof. Directly from the Soundness and Completeness theorems and Corollary 7.1.1. □

7.4 Summary and Related Work

In this chapter we studied the symbolic input-output relation of `utcc` processes to give a more concrete representation of the actual output of a process. We defined this relation as the set of pairs (w, v) such that v is the least sequence entailing the same information (basic constraints) than v' in $P \xrightarrow{(w,v')}_s$. We showed that this relation is a closure operator for the monotonic fragment of the calculus and then, it can be fully characterized by its set of fixed points here called the strongest postcondition.

Following the semantics for CCP and `tcc` in [Saraswat 1991, de Boer 1995b, Saraswat 1994, Nielsen 2002a], we gave a compositional characterization of the strongest postcondition relation. Since the symbolic semantics outputs past-monotonic sequences, in our denotational model, the codomain was defined as sequences of past-monotonic sequences and not as sequences of basic constraints as in `tcc`.

We proved our denotational model to be fully abstract with respect to the symbolic semantics for the monotonic fragment. This then allowed us to retrieve compositionally the input-output behavior of monotonic processes. We shall use this result in Chapter 8 where we give a semantic account based on closure operators to a language for security protocols based on the semantics here presented.

The material of this chapter was originally published as [Olarte 2008b].

Chapter 8

Closure Operators for Security

Due to technological advances such as the Internet and mobile computing, *security* has become a serious challenge involving several fields of Computer Science, in particular Process Calculi. Typically, these calculi provide mechanisms for communication of private names (*nonces*), i.e., mobility as is understood in this dissertation. Furthermore, they offer a set of reasoning techniques to verify if a given cryptographic property such as *secrecy* or *authentication* holds.

Remarkably, most process calculi for security protocols have strong similarities with CCP. For instance, SPL [Crazzolara 2001], the Spi calculus variants in [Abadi 1997] and [Amadio 2003], and the calculus in [Boreale 2000] are all operationally defined in terms of configurations containing information which can only increase during evolution. Such a monotonic evolution of information is akin to the notion of *monotonic store*, which is central to CCP and a source of its simplicity. Also, the calculi in [Amadio 2003, Fournet 2003, Boreale 2001a] are parametric in the underlying logic much like CCP is parametric in an underlying constraint system.

In this chapter we show how the monotonic fragment of `utcc` and its closure operator characterization can be used to give meaning to a SPL-like process language enjoying the typical features of calculi for security mentioned above. This language, called SCCP (Security Concurrent Constraint Programming Language), arises as a specialization of `utcc` with a particular cryptographic constraint systems. We shall show that processes in the language can be compositionally specified as closure operators. This way, the set of messages a protocol may produce can be represented as a closure operator over sequences of temporal constraints (i.e., future free formulae).

We believe that the interpretation of the behavior of protocols as closure operators is a natural one. For instance, a spy can only produce new information (extensiveness); the more information she gets, the more she will infer (monotonicity); and she infers as much as possible for the information she gets (idempotence). To our knowledge no closure operator denotational account has previously been given in the context of calculi for security protocols. We then bring new semantic insights into the modeling and verification of security protocols.

Finally, in this chapter we also show that the declarative characterization of `utcc` processes as FLTL formulae allows for reachability analysis of security protocols modeled in SCCP. In particular, we can verify if a protocol may reach a state where a secrecy property is violated.

8.1 The modeling language: SCCP

As a modeling language, we shall use a syntax of processes following that of the *Security Protocol Language* (SPL) defined in [Crazzolara 2001]. Roughly speaking, this language offers primitives to output and receive messages as well as to generate secrets or nonces (randomly-generated unguessable items). We shall refer to this language as SCCP (Security Concurrent Constraint Programming Language).

Definition 8.1.1 (Syntax of SCCP)**.** *The SCCP language is given by the following syntax:*

$$
\begin{array}{llll}
\textit{Values} & v, v' & ::= & n \mid x \\
\textit{Keys} & k & ::= & \mathit{pub}(v) \mid \mathit{priv}(v) \\
\textit{Messages} & M, N & ::= & v \mid k \mid (M, N) \mid \{M\}_k \\
\textit{Processes} & R, R' & ::= & \mathbf{nil} \\
 & & & \mid \mathbf{new}(x)R \\
 & & & \mid \mathbf{out}(M).R \\
 & & & \mid \mathbf{in}\ (\vec{x})[M].R \\
 & & & \mid\ !R \\
 & & & \mid R \parallel R'
\end{array}
$$

SCCP includes a set of names (values) with n, m, A, B ranging over it. These values represent ids of principals $(A, B, ...)$ or nonces $(n, m, ...)$. The set of keys is built upon two constructors providing the public ($\mathit{pub}(v)$) and the private key ($\mathit{priv}(v)$) associated to a value.

Messages can be constructed from composition (M, N) and encryption $\{M\}_k$. As explained below, message decomposition and decryption can be obtained by using pattern matching on inputs.

8.1.1 Processes in SCCP

Intuitively, processes in SCCP run in time intervals. This way, a process of the form $P.R$ represents a process executing P in the current time interval and R in the next one.

The output process $\mathbf{out}(M).R$ broadcasts M over the network and then it behaves as R in the next time unit. As standardly done, messages are supposed to be sent to an untrusted network where the spy can see and store all of them (see e.g., [Fiore 2001]).

The input $\mathbf{in}\ (\vec{x})[M].R$ waits for all the messages of the form $M[\vec{t}/\vec{x}]$ to be output on the network and then behaves like $R[\vec{t}/\vec{x}]$ in the next time unit. This process binds the variables $\vec{x}$ in R. For example, if the message $(A, B)_{pub(k)}$ is ouput, the process $\mathbf{in}\ (x, y)[(x, y)_{pub\,(k)}].R$ executes $R[A/x, B/y]$ in the next time unit.

The process $\mathbf{new}(x)R$ generates a (nonce) x private to R. The process $\mathbf{nil}$ does nothing and $R \parallel R'$ denotes the parallel execution of R and R'. Given a finite set of indexes $I = \{i_1, i_2, ..., i_n\}$, we shall use $\prod_{i \in I} P_i$ to denote the parallel composition $P_{i_1} \parallel P_{i_2} \parallel ... \parallel P_{i_n}$.

Finally $!\,R$ denotes the execution of R in each time unit.

8.2 Dolev-Yao Constraint System

Typically, in the modeling of security protocols one must take into account all possible actions the attacker may perform. This attacker is usually given in terms of the Dolev and Yao thread model [Dolev 1983] which presupposes an attacker that can eavesdrop, disassemble, compose, encrypt and decrypt messages with available keys. It also presupposes that cryptography is unbreakable.

Before giving a closure operator semantics to our security language, we then need a constraint system handling the cryptographic constructs (e.g., message encryption and composition) and whose entailment relation follows the inferences a Dolev-Yao attacker may perform.

Definition 8.2.1. *Let Σ_s be a signature with constant symbols in $\mathcal{V}$, function symbols enc, pair, priv and pub and the unary predicate* `out`*. Let Δ_s be the closure under deduction of $\{\ F \mid \vdash_s F\ \}$ with $\vdash_s$ as in Table 8.1. The (secure) constraint system is the pair (Σ_s, Δ_s).*

$$\text{PRJ}\ \frac{F \vdash_{\mathtt{s}} \mathtt{out}((m_1, m_2))}{F \vdash_{\mathtt{s}} \mathtt{out}(m_i) \qquad i \in \{1,2\}}$$

$$\text{PAIR}\ \frac{F \vdash_{\mathtt{s}} \mathtt{out}(m_1) \quad F \vdash_{\mathtt{s}} \mathtt{out}(m_2)}{F \vdash_{\mathtt{s}} \mathtt{out}((m_1, m_2))}$$

$$\text{ENC}\ \frac{F \vdash_{\mathtt{s}} \mathtt{out}(m) \quad F \vdash_{\mathtt{s}} \mathtt{out}(k)}{F \vdash_{\mathtt{s}} \mathtt{out}(\{m\}_{pub(k)})}$$

$$\text{DEC}\ \frac{F \vdash_{\mathtt{s}} \mathtt{out}(priv(k)) \quad F \vdash_{\mathtt{s}} \mathtt{out}(\{m\}_{pub(k)})}{F \vdash_{\mathtt{s}} \mathtt{out}(m)}$$

Table 8.1: Security constraint system entailment relation.

Intuitively, $\mathcal{V}$ represents the set of principal ids, nonces and values. We use $\{m\}_k$ and (m_1, m_2) respectively, for $enc(m, k)$ (encryption) and $pair(m_1, m_2)$ (composition).

Rule PRJ in Table 8.1 says that if one can infer a composed message (m_1, m_2), then one can infer also the components of the message, i.e., m_1 and m_2. Rule PAIR is the converse of the previous one: given two messages m_1 and m_2, one can infer the composition (m_1, m_2). Rule ENC says that if one can infer that the message m as well as a key k are output on the global channel $\mathtt{out}$, then one may as well infer that $\{m\}_{pub(k)}$ is also output on $\mathtt{out}$. Finally, DEC dictates that the message m can be deduced if both, the encrypted message $\{m\}_{pub(k)}$ and the corresponding private key $priv(k)$ can be deduced.

Let us note two important issues about the constraint system above.

Remark 8.2.1. *Firstly, notice that the secure constraint system in Definition 8.2.1 introduces infinitely many internal reductions if we were to observe the behavior of a process by using the operational semantics. To see this, assume that* $F \vdash_s \mathtt{out}(m)$ *and let* $P = (\mathbf{abs}\ x; \mathtt{out}(x))\, P'$. *Then, from* F *it is also possible to entail* $\mathtt{out}(m, m)$, $\mathtt{out}(m, (m, m))$, *etc. Then,* P *must execute* P' *for all these possible terms.*

Secondly, we note that for the sake of presentation we added the capabilities of the spy as inferences rules in the constraint system. Nevertheless, those rules can be easily specified as `utcc` *processes, thus leading to a simpler constraint system with the empty theory* ($\Delta = \emptyset$). *Take for example the rule* ENC. *One can define this ability of the spy as the process* $P_{Enc} = (\mathbf{abs}\ x, k; \mathtt{out}(x) \wedge \mathtt{out}(k))\, \mathbf{tell}(\mathtt{out}(\{x\}_{pub(k)}))$.

8.3 Modeling a Security Protocol in SCCP

To illustrate the language SCCP, consider the Needham-Schröder (NS) protocol described in [Needham 1978]. This protocol aims at distributing two *nonces* in a secure way, whose purpose is to ensure the freshness of messages.

Figure 8.1(a) shows the steps of NS where m and n represent the nonces generated, respectively, by the principals A and B. The protocol initiates when A sends to B a new nonce m together with her own agent name A, both encrypted with B's public key. When B receives the message, he decrypts it with his secret private key. Once decrypted, B

M_1	$A \to B$	:	$\{(m, A)\}_{pub(B)}$	M_1	$A \to C$	:	$\{(m, A)\}_{pub\,(C)}$
M_2	$B \to A$	:	$\{(m, n, B)\}_{pub(A)}$	M'_1	$C \to B$	:	$\{(m, A)\}_{pub\,(B)}$
M_3	$A \to B$	:	$\{n\}_{pub(B)}$	M_2	$B \to A$	:	$\{(m, n, B)\}_{pub\,(A)}$
	(a)			M_3	$A \to C$	:	$\{n\}_{pub\,(C)}$
					(b)		

Figure 8.1: Needham-Schroeder Protocol

prepares an encrypted message for A that contains a new nonce together with the nonce received from A and his name B. Acting as responder, B sends it to A, who recovers the clear text using her private key. A convinces herself that this message really comes from B by checking whether she got back the same nonce sent out in the first message. If that is the case, she acknowledges B by returning his nonce. B does a similar test.

Secrecy Attack. Assume the execution of the protocol between A, B and C in Figure 8.1(b). Here C is an intruder, i.e. a malicious agent playing the role of a principal in the protocol. As it was shown in [Lowe 1996], this execution leads to a secrecy flaw where the attacker C can reveal n which is meant to be known only by A and B.

In this execution, the attacker replies to B the message sent by A and B believes that he is establishing a session key with A. Since the attacker knows the nonce m from the first message, he can decrypt the message $\{n\}_{pub\,(C)}$ and n is not longer a secret between A and B as intended.

The Model in SCCP. We model the behavior of the initiator and the responder in our running example as follows:

$$
\begin{aligned}
\mathsf{Init}(A, B) \equiv\ & !\,\mathbf{new}(m) \\
& \quad \mathbf{out}(\{(m, A)\}_{pub(B)}). \\
& \quad\ \ \mathbf{in}\ (x)[\{(m, x, B)\}_{pub(A)}].\mathbf{out}(\{x\}_{pub(B)}).\mathbf{nil}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Resp}(\mathsf{B}) \equiv\ & !\,\mathbf{in}\ (x, u)[\{(x, u)\}_{pub(B)}]. \\
& \quad \mathbf{new}(n) \\
& \quad\ \ \mathbf{out}(\{m, n, B\}_{pub\,(u)}).\mathbf{nil}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Spy} \equiv\ & \|_{A \in \mathcal{P}}\ !\,\mathbf{out}(A).\mathbf{nil} \\
& \|_{A \in \mathcal{P}}\ !\,\mathbf{out}(pub(A)).\mathbf{nil} \\
& \|_{A \in Bad}\ !\,\mathbf{out}(priv(A)).\mathbf{nil}
\end{aligned}
$$

The process Spy corresponds to the initial knowledge the attacker has. Given the set of principals of the protocol $\mathcal{P}$, the spy knows all the names of the principals in the protocol and their public keys. He also knows a set of private keys denoted by Bad. This set represents the leaked keys, for example, the private key of C in the above configuration exhibiting the secrecy flaw (Figure 8.1 (b)).

Notice that the processes Init and Resp are replicated. This models the fact that principal may initiate different sessions during the execution of the protocol.

8.4 Closure Operator semantics for SCCP

In this section we give a closure operator semantics to SCCP following that of `utcc` in Chapter 7.

We start by defining a compositional mapping from SCCP constructs into monotonic `utcc` processes.

Definition 8.4.1. *Let I be a function from SCCP to monotonic* `utcc` *processes defined by:*

$$I(P) = \begin{cases} \mathbf{skip} & \text{if } R = \mathbf{nil} \\ (\mathbf{local}\, x)\, I(R') & \text{if } R = \mathbf{new}(x)R' \\ !\,\mathbf{tell}(\mathtt{out}(M)) \parallel \mathbf{next}\, I(R') & \text{if } R = \mathbf{out}(M).R' \\ (\mathbf{abs}\ \vec{x}; \mathtt{out}(M))\, \mathbf{next}\, I(R') & \text{if } R = \mathbf{in}\ (\vec{x})[M].R' \\ \prod_{i \in I} I(R_i) & \text{if } R = \prod_{i \in I} R_i \\ !\, I(R') & \text{if } R =! R' \end{cases}$$

It is easy to see that the above interpretation realizes the behavioral intuition of SCCP given before. Intuitively the output $\mathbf{out}(M)$ is mapped to a process adding the constraint $\mathtt{out}\,(M)$. Since the final store in `utcc` is not automatically transferred to the next time interval, the process $\mathbf{tell}(\mathtt{out}(M))$ is replicated. This reflects also the fact that the attacker can remember all the messages posted over the network.

For the case of the input process $\mathbf{in}\ (\vec{x})[M].R'$, we use an abstraction to execute the process $\mathbf{next}\, R'[\vec{t}/\vec{x}]$ for every message of the form $M[\vec{t}/\vec{x}]$ output on the network, i.e., when a constraint of the form $\mathtt{out}(M)[\vec{t}/\vec{x}]$ can be deduced.

Semantics of SCCP The following function maps our security processes into its set of fixed points as specified in Table 7.1—i.e., its strongest postcondition.

Definition 8.4.2. *For any SCCP process R we define $[\![R]\!]_{SCCP}$ as $[\![I(R)]\!]$ with $I(\cdot)$ as in Definition 8.4.1 and $[\![\cdot]\!]$ as in Table 7.1.*

Since the interpretation function I is given in terms of the monotonic fragment of `utcc`, it follows from Section 7.1 that $[\![R]\!]_{SCCP}$ corresponds to a closure operator.

8.4.1 Closure Properties of SCCP

We conclude this section by pointing out that the interpretation of the behavior of protocols as closure operators is a natural one. To see our intuition, let us suppose that f is a closure operator denoting a SCCP Spy eavesdropping and producing information in the network. Assume also that w, v are sequences of constraints representing the set of messages posted on the network, i.e., the information available from the execution of the protocol.

- **Extensiveness** $f(w) \succeq w$: The Spy produces new information from the one he obtains.
- **Monotonicity**. If $w \succeq v$ then $f(w) \succeq f(v)$: The more information the Spy gets, the more he will infer.
- **Idempotence** $f(f(w)) = f(w)$: The Spy infers as much as possible from the info he gets.

8.5 Verification of Secrecy Properties

We can represent protocols in such a way that potential attacks can be specified as the least fixed point of the closure operators representing them. To detect when the secret created by Resp is revealed by the attacker, we modify the definition of this process as follows:

$$\begin{aligned}\mathsf{Resp}'(B) \equiv \quad & \mathbf{in}\ (x,u)[\{(x,u)\}_{pub(B)}].\\ & \mathbf{new}(n)(\mathbf{out}(\{m,n,B\}_{pub\,(u)}).\mathbf{nil}) \parallel\\ & \quad !\,\mathbf{in}\ [n].\mathbf{out}(attack).\mathbf{nil})\end{aligned}$$

Intuitively, Resp' outputs the message *attack* when the message n appears unencrypted on the network, i.e., when $\mathtt{out}(n)$ can be deduced from the current store.

Recall that `false` is an absorbing element for conjunction and it is the greatest future-free formula with respect to $\succeq$. Our approach is then to add the constraint `false` when the message *attack* can be read from the network. In terms of the closure operator semantics it implies that if a process R outputs the message *attack*, then every fixed point of the closure operator representing R is a sequence whose suffix is the sequence $\mathtt{false}^\omega$. More precisely,

Proposition 8.5.1. *Let R be a SCCP process. Let f be defined as*

$$f = [\![R]\!]_{SCCP} \cap [\![!\,\mathbf{when}\ \mathtt{out}(attack)\ \mathbf{do}\ !\,\mathbf{tell}(\mathtt{false})]\!]$$

Therefore, $I(R) \Downarrow_s^{attack}$ iff all fixed point of the closure operator corresponding to f takes the form $w.\,\mathtt{false}^\omega$

Proof. Immediate from the definition of f and full-abstraction in Theorem 7.3.3.

□

The previous proposition allows us to exhibit the secrecy flaw in our running example. Let $\mathcal{P} = \{A, B, C\}$ be the set of principal names and $Bad = \{C\}$ be the set of leaked keys in our previous protocol example. Given the process

$$\mathsf{NS} = \mathsf{Init}(A,C) \parallel \prod_{X\in\mathcal{P}} \mathsf{Resp}'(X) \parallel \mathsf{Spy}$$

and $f = [\![\mathsf{NS}]\!]_{SCCP} \cap [\![!\,\mathbf{when}\ \mathtt{out}(attack)\ \mathbf{do}\ !\mathbf{tell}(\mathtt{false})]\!]$, one can verify that the least fixed point v of f takes the form $v = w.\,\mathtt{false}^\omega$.

8.6 Reachability Analysis in SCCP

In the previous section we used the semantic characterization of `utcc` processes as closure operators to exhibit a secrecy flaw in a security protocol. In this section we show how the declarative characterization of `utcc` as FLTL formulae can be exploited to perform reachability analysis of a security protocol modeled in SCCP.

Remind that the process Resp' outputs the constraint *attack* when the nonce generated by the responder has been sent on the global channel `out`. Then, by appealing to the FLTL correspondence in Theorem 5.2.3 and Proposition 8.5.1 we have:

Proposition 8.6.1. *Let R be a SCCP process and I as in Definition 8.4.1. Let $A = \mathsf{TL}[\![I(R)]\!]$ be the FLTL formula corresponding to $I(R)$ (Definition 5.1.1) and*

$$f = [\![R]\!]_{SCCP} \cap [\![!\,\mathbf{when}\ \mathtt{out}(attack)\ \mathbf{do}\ !\,\mathbf{tell}(\mathtt{false})]\!]$$

The following statements are equivalent

- ***Symbolic Output***: $I(R) \Downarrow_s^{attack}$
- ***Closure Operator Semantics***: *The least-fixed point of the closure operator corresponding to* f *takes the form* $w.\texttt{false}^\omega$
- ***FLTL Characterization***: *There exists* $k \geq 0$ *such that* $Cut_F(A,k) \models_T \Diamond attack$

Proof. Directly from Theorems 5.2.3 and 7.3.3 and the definition of f. □

8.7 Summary and Related Work

In this chapter we defined SCCP, a simple language for the specification of security protocols based on Crazzolara and Winskel's SPL [Crazzolara 2001]. This language arises as a specialization of `utcc` with a particular cryptographic constraint system and features mechanisms to create local names (nonces) and to specify input and output of messages on a network. We made use of the semantics in Chapter 7 to give a closure operator interpretation of this language. We then showed that the least fixed point of the closure operator associated to the process modeling a protocol may allow us to verify whether a *secrecy* property is not verified. Furthermore, relying on the FLTL characterization of `utcc` processes in Chapter 5, we showed that it is possible to verify if a protocol reaches a state where a secrecy property is violated. We illustrate our approach by exhibiting a well known attack in the Needham-Schroeder protocol [Needham 1978] described in [Lowe 1996].

The material of this chapter was originally published as [Olarte 2008c, Olarte 2008b].

Related Work. Several process languages have been defined to analyze security protocols. For instance Crazzolara and Winskel's SPL [Crazzolara 2001], the spi calculus variants by Abadi [Abadi 1997] and Amadio [Amadio 2003], Boreale's calculus in [Boreale 2000] and the Applied π-calculus [Fournet 2003] among others. Although `utcc` can be used to reason about certain aspects of security protocols (e.g., secrecy), it was not specifically designed for this application domain. Here we illustrated how the closure operator semantics of `utcc` may offer new reasoning techniques for the verification of security protocols. We also argued for the closure operators as a natural characterization of the information that can be inferred (e.g., by Spy) from a protocol. To our knowledge closure operators had not been considered in the study of security protocols.

The successful *logic programming* approach to security protocols in [Abadi 2005] is closely related to ours. Basically, in [Abadi 2005] protocols are modeled as a set of Horn clauses rather than processes. The verification of the secrecy property relies in deducing (or proving that it is not possible) the predicate $attack(M)$ from the set of Horn clauses. A benefit from our approach is that we can overcome the problem of false attacks pointed in [Blanchet 2005]: Consider for example a piece of data that needs to be kept secret in a first phase of the protocol and later is revealed when it is not required to be a secret. Because the lack of temporal dependency this may generate a false attack. The temporal approach here presented may allow us to control when a message is required to be secret. The work in [Blanchet 2005] also avoid false attacks by using a linear logic [Girard 1987] approach rather than a temporal one.

The authors in [Hildebrandt 2009] show that the abstraction operator in `utcc` allows agents to guess channel names and encrypted values by universal quantification. For example, the process $(\textbf{abs}\ x, y; \texttt{out}(x,y))\ P$ is able to capture all possible messages in transit due to the quantification of the channel name x. Furthermore, a process of the form

(**abs** $x, k; \mathtt{out}(\{x\}_{pub(k)}))\, P$ is able to reveal the message x without knowing the private key associated to k. The authors then propose a type system to guarantee that, e.g., channel names and encrypted values are only extracted by agents that are able to infer the channel or the nonencrypted value from the store.

The model of the attacker in our secrecy analysis is given by the inference rules of the cryptographic constraint system in Definition 8.2.1. This means, the attacker is not represented as an arbitrary process running in parallel with the specification of the protocol. Thus, our model of the attacker can only deduce new information according to the Dolev and Yao thread model [Dolev 1983]. Aiming at developing reasoning techniques for `utcc` based on behavioral equivalences like those in [Abadi 1997, Fournet 2003], must certainly take into account the work in [Hildebrandt 2009] to rule out contexts defining a spy "more powerful" than a Dolev-Yao attacker.

Finally, in Chapter 10 we shall introduce an abstract semantics for `utcc` that approximates the semantics in Chapter 7. We then show that the fixed point of the abstract semantics can be computed in a finite number of steps. This way, with the help of prototypical implementation, we shall exhibit the secrecy flaw illustrated in this chapter automatically.

Chapter 9

Applications

We have illustrated in the previous chapters the applicability of the utcc calculus in the modeling and verification of security protocols. It is worth noticing that utcc was not specifically designed for this application domain but to model in general mobile reactive systems. In this chapter we show that utcc has much to offer in the specification and verification of systems in two emergent application areas. Namely, Service Oriented Computing and Multimedia Interaction Systems.

Service Oriented Computing. In Section 9.1, we shall give an alternative interpretation of the π-based language defined by Honda, Vasconcelos and Kubo, henceforth referred to as HVK, for structured communications [Honda 1998]. The encoding of HVK into utcc is straightforwardly extended to explicitly model information on session duration, allows for declarative preconditions within session establishment constructs, and features a construct for session abortion. Then, a richer language for the analysis of sessions is defined where time can be explicitly modeled. Additionally, relying on the FLTL characterization of utcc processes as FLTL formulae, reachability analysis of sessions can be characterized as FLTL entailment.

Multimedia Interaction Systems. As second application domain, in Section 9.5 we argue for utcc as a language for the modeling of dynamic multimedia interaction systems. We shall show that the notion of constraints as partial information allows us to neatly define temporal relations between interactive agents or events. Furthermore, mobility in utcc allows for the specification of more flexible and expressive systems in this setting, thus broadening the interaction mechanisms available in previous models.

9.1 Service Oriented Computing

Service Oriented Computing (or SOC for short) is often seen as a natural progression from component based software development, and as a mean to integrate different component development frameworks. A service in this context may be defined as a behavior that is provided by a component to be used by any other component. This behavior is described by an interface contract identifying the capabilities provided by the service.

SOC is in principle different from distributed systems as it gives more abilities to the agents involved. First of all, services are composable by nature, meaning that a service can be either a singular activity or a process where different services are assembled to provide a result. Second, service composition is open; in terms that any service that matches the requirements specified by the service requester should be able to be bound in the composition. Third, services can be loosely-coupled, referring to the capability of a service to interact in ever-changing environments, probably moving its partial computations to different locations where services are more reliable; Finally, services can operate in asynchronous environments, where a single computation can take months or even years to execute.

Modeling Services. From the viewpoint of *reasoning techniques*, two main trends in modeling in SOC can be singled out. On the one hand, a *behavioral approach* focuses on how process interactions can lead to correct configurations. Typical representatives of this approach are based on process calculi and Petri nets (see, e.g., [Lapadula 2007, Boreale 2006, Lanese 2007]), and count with behavioral equivalences and type disciplines as main analytic tools. On the other hand, in a *declarative approach* the focus is on the set of conditions components should fulfill in order to be considered correct, rather than on the complete specification of the control flows within process activities (see, e.g., [Pesic 2006]). Even if these two trends address similar concerns, they have evolved rather independently from each other.

We shall show that `utcc` may allow for an approach in which behavioral and declarative techniques can harmoniously converge for the analysis of sessions. More precisely, we shall give an alternative interpretation to the language defined by Honda, Vasconcelos and Kubo in [Honda 1998] (`HVK`). This way, structured communications can be studied in a declarative framework in which time is explicit. We begin by proposing an encoding of the `HVK` language into `utcc`; such an encoding defines asynchronous session establishment and satisfies a rather standard operational correspondence property. We then move to the timed setting, and propose `HVK-T`, a timed extension of the `HVK` language. The extended language explicitly includes information on session duration, allows for declarative preconditions within session establishment constructs, and features a construct for session abortion. We then show that the encoding of `HVK` into `utcc` straightforwardly extends to `HVK-T`.

9.2 A Language for Structured Communication

We begin by introducing `HVK`, the language for structured communication proposed in [Honda 1998]. We assume the following notational conventions: *names* are ranged over by $a, b, \ldots$; *channels* are ranged over by k, k'; *variables* are ranged over by $x, y, \ldots$; *constants* (names, integers, booleans) are ranged over by $c, c', \ldots$; *expressions* (including constants) are ranged over by $e, e', \ldots$; *labels* are ranged over by $l, l', \ldots$; *process variables* are ranged over by $X, Y, \ldots$. Finally, $u, u', \ldots$ denote names and channels. The sets of free names/channels/variables/process variables of P, is defined in the standard way, and respectively denoted by $fn(\cdot), fc(\cdot), fv(\cdot)$ and $fpv(\cdot)$. Processes without free variables or free channels are called *programs*.

Definition 9.2.1 (The `HVK` language [Honda 1998])**.** *Processes in `HVK` are built from:*

P, Q	::=	**request** $a(k)$ **in** P	*Session Request*
	|	**accept** $a(x)$ **in** P	*Session Acceptance*
	|	$k![\vec{e}];\ P$	*Data Sending*
	|	$k?(x)$ **in** P	*Data Reception*
	|	$k \lhd l; P$	*Label Selection*
	|	$k \rhd \{l_1 : P_1 \parallel \cdots \parallel l_n : P_n\}$	*Label Branching*
	|	**throw** $k[k'];\ P$	*Channel Sending*
	|	**catch** $k(k')$ **in** P	*Channel Reception*
	|	**if** e **then** P **else** Q	*Conditional Statement*
	|	$P \mid Q$	*Parallel Composition*
	|	**inact**	*Inaction*
	|	$(\nu u)P$	*Hiding*
	|	**def** D **in** P	*Recursion*
	|	$X[\vec{e}\,\vec{k}]$	*Process Variables*

$$D \quad ::= \quad X_1(x_1k_1) = P_1 \textbf{ and} \cdots \textbf{and } X_n(x_nk_n) = P_n$$

9.2.1 Operational Semantics of HVK

The operational semantics of HVK is given by the reduction relation $\longrightarrow_{\texttt{HVK}}$ which is the smallest relation on processes generated by the rules in Table 9.2. In Rule Str, the structural congruence $\equiv$ is the smallest relation satisfying :

1. $P \equiv Q$ if they differ only by a renaming of bound variables (alpha-conversion).
2. $P \mid \textbf{inact} \equiv P$, $P \mid Q \equiv Q \mid P$, $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$.
3. $(\nu u)\textbf{inact} \equiv \textbf{inact}$, $(\nu uu)P \equiv (\nu u)P$, $(\nu uu')P \equiv (\nu u'u)P$, $(\nu u)(P \mid Q) \equiv (\nu u)P \mid Q$ if $x \notin fv(Q)$, $(\nu u)(\textbf{def } D \textbf{ in } P) \equiv (\textbf{def } D \textbf{ in } ((\nu u)P))$ if $u \notin fv(D)$.
4. $(\textbf{def } D \textbf{ in } P) \mid Q \equiv \textbf{def } D \textbf{ in } (P \mid Q)$ if $fpv(D) \cap fpv(Q) = \emptyset$.
5. $\textbf{def } D \textbf{ in } (\textbf{def } D' \textbf{ in } P) \equiv \textbf{def } D$ and $D' \textbf{ in } P$ if $fpv(D) \cap fpv(D') = \emptyset$.

Link	$\textbf{accept } a(x) \textbf{ in } P \mid \textbf{request } a(k) \textbf{ in } Q \longrightarrow_{\texttt{HVK}} (\nu k)(P \mid Q)$
Com	$(k![\vec{e}]; P) \mid (k?(x) \textbf{ in } Q) \longrightarrow_{\texttt{HVK}} P \mid Q[\vec{c}/\vec{x}]$ if $e \downarrow \vec{c}$
Label	$k \lhd l_i; P \mid k \rhd \{l_1 : P_1 \parallel \cdots \parallel l_n : P_n\} \longrightarrow_{\texttt{HVK}} P \mid Pi \ (1 \leq i \leq n)$
Pass	$\textbf{throw } k[k']; P \mid \textbf{catch } k(k') \textbf{ in } Q \longrightarrow_{\texttt{HVK}} P \mid Q$
If1	$\textbf{if } e \textbf{ then } P \textbf{ else } Q \longrightarrow_{\texttt{HVK}} P \ (e \downarrow \texttt{true})$
If2	$\textbf{if } e \textbf{ then } P \textbf{ else } Q \longrightarrow_{\texttt{HVK}} Q \ (e \downarrow \texttt{false})$
Def	$\textbf{def } D \textbf{ in } (X[\vec{e}\vec{k}] \mid Q) \longrightarrow_{\texttt{HVK}} \textbf{def } D \textbf{ in } (P[\vec{c}/\vec{x}] \mid Q) \ (e \downarrow \vec{c}, X(\vec{x}\vec{k}) = P \in D)$
Scop	$P \longrightarrow_{\texttt{HVK}} P'$ implies $(\nu u)P \longrightarrow_{\texttt{HVK}} (\nu u)P'$
Par	$P \longrightarrow_{\texttt{HVK}} P'$ implies $P \mid Q \longrightarrow_{\texttt{HVK}} P' \mid Q$
Str	If $P \equiv P'$ and $P' \longrightarrow_{\texttt{HVK}} Q'$ and $Q' \equiv Q$ then $P \longrightarrow_{\texttt{HVK}} Q$

Table 9.2: Reduction Relation for HVK [Honda 1998].

Let us give an intuition about the rules above. The central idea in HVK is the notion of session. A session is a series of reciprocal interactions between two parties, possibly with branching and recursion, and serves as a unit of abstraction for describing interaction.

Communications belonging to a session are done via a port specific to that session, called a channel which is generated when initiating each session.

The initialization of a session in HVK can be specified by a process of the form

$$\textbf{accept } a(x) \textbf{ in } P \mid \textbf{request } a(k) \textbf{ in } Q$$

In this case, the request first requests, via a name a, the initiation of a session as well as the generation of a fresh channel. The accept, on the other hand, receives the request for the initiation of a session via a, and generates a new channel k which is used for P and Q to communicate each other.

Three kinds of atomic interactions are available in the language: sending (including name passing), branching and channel passing (or delegation). Those actions are described

$$
\begin{array}{lcl}
ATM(a,b) & = & \textbf{accept}\ a(k)\ \textbf{in}\ k?(id)\ \textbf{in} \\
& & k \rhd \left\{ \begin{array}{l}
deposit : \textbf{request}\ b(h)\ \textbf{in} \\
\quad k?(amt)\ \textbf{in}\ h \lhd deposit; \\
\quad h![id, amt]; ATM(a,b) \\
||withdraw : \textbf{request}\ b(h)\ \textbf{in} \\
\quad k?(amt)\ \textbf{in}\ h \lhd withdraw; h![id, amt]; \\
\quad h \rhd \left\{ \begin{array}{l} success : k \lhd dispense; k![amt]; ATM(a,b) \\ ||failure : k \lhd overdraft; ATM(a,b) \end{array} \right\} \\
||balance : \textbf{request}\ b(h)\ \textbf{in}\ h \lhd balance; h?(amt)\ \textbf{in} \\
\quad k![amt]; ATM(a,b)
\end{array} \right\}
\end{array}
$$

Figure 9.1: ATM process specification [Honda 1998]

by the rules Com, Label and Pass respectively. In the case $(k![\vec{e}]; P) \mid (k?(x)\ \textbf{in}\ Q)$, the expression $\vec{e}$ is sent on the port (session channel) k. Then the process $k?(x)\ \textbf{in}\ Q$ receives such a data and then execute $Q[\vec{c}/\vec{x}]$ where $\vec{c}$ is the result of evaluating the expression $\vec{e}$.

The case of Pass is similar but considering that only session names are transmitted in $\textbf{throw}\ k[k']; P \mid \textbf{catch}\ k(k')\ \textbf{in}\ Q$.

In the case of $k \lhd l_i; P \mid k \rhd \{l_1 : P_1 \parallel \cdots \parallel l_n : P_n\}$, the process $k \lhd l_i; P$ selects one label and then the corresponding process P_i is executed.

The other rules are self-explanatory.

9.2.2 An Example

Let us give an intuition on how a declarative approach could be useful in the analysis of sessions. Consider the ATM example from [Honda 1998, Section 4.1] (see Figure 9.1). There, an ATM has established sessions with a user (via the channel a) and his bank (via the channel b). it allows for `deposit`, `balance`, and `withdraw` operations. In the latter case, if there is not enough money to withdraw, then an *overdraft* message appears to the user.

Consider now an extension of the previous system with a malicious card reader that keeps the user's sensible information and uses it to continue withdrawing money without his/her authorization. A greedy card reader could even repeatedly withdraw until causing an overdraft, as in Figure 9.2.

The card reader acts as an interface between the user and the ATM. By creating sessions between them, the card reader is able to receive the data about the user's identity and use that data later to establish a session with the ATM. The card reader then uses the information associated to the user's transaction to first provide him the money and then to continue withdrawing more money without the user's authorization; this is the role of recursive process R. The process Q above can be assumed to be a process that sends a message through a session with the bank saying that the account has run out of money: $Q = k_{bank}![\underline{0}]; \textbf{inact}$.

In this simple scenario, the correspondence between `utcc` and first-order linear-time temporal logic (FLTL) may come in handy to reason about the possible states for this specification. These can be used not only to describe the operational behavior of the compromised ATM above, but also to provide declarative arguments regarding its evolution. For instance, assuming Q as above, one could show that a `utcc` specification of the ATM example satisfies the FLTL formula $\Diamond\, \texttt{out}(k_{bank}, 0)$, which intuitively means that in the presence of the malicious card reader the user's bank account will eventually go to overdraft.

$$
\begin{array}{lcl}
\mathit{Reader} & = & \textbf{accept}\ r(k')\ \textbf{in}\ k'?(id)\ \textbf{in} \\
 & & \quad \textbf{request}\ a(k)\ \textbf{in}\ k![id]; \\
 & & \quad k' \rhd \left\{ \begin{array}{l} \mathit{withdraw} : k'?(amt)\ \textbf{in} \\ k \lhd \mathit{withdraw}; k![amt]; \\ \quad k \rhd \left\{ \begin{array}{l} \mathit{dispense} : k' \lhd \mathit{dispense}; k![amt]; R(k, amt) \\ \parallel\ \mathit{overdraft} : Q \end{array} \right\} \end{array} \right\} \\
R(j,x) & = & \textbf{def}\ R'\ \textbf{in}\ k \lhd \mathit{withdraw}; j![x]; j \rhd \left\{ \begin{array}{l} \mathit{dispense} : j?(amt)\ \textbf{in}\ R(j,x) \\ \parallel\ \mathit{overdraft} : Q \end{array} \right\} \\
\mathit{User} & = & \textbf{request}\ r(k')\ \textbf{in}\ k'![myId]; \\
 & & \quad k' \lhd \mathit{withdraw}; k'![58]; \quad k' \rhd \{\mathit{dispense} : k'?(amt)\ \textbf{in}\ P \parallel \mathit{overdraft} : Q\}
\end{array}
$$

Figure 9.2: ATM Example with a Malicious Card Reader.

9.3 A Declarative Interpretation for Sessions

The Table 9.4 presents a compositional encoding of HVK processes into utcc. In this encoding we make us of the derived constructs ($\underline{\textbf{wait}}\ \vec{x}; c$) **do** Q and $\underline{\textbf{tell}}(c)$ that we defined in Section 6.5.3. Recall that ($\underline{\textbf{wait}}\ \vec{x}; c$) **do** Q is a persistent abstraction waiting for possibly several time units until for some $\vec{t}$, $c[\vec{t}/\vec{x}]$ holds. Then it executes $Q[\vec{t}/\vec{x}]$. The process $\underline{\textbf{tell}}(c)$ outputs the constraint c in several time units until a process of the form ($\underline{\textbf{wait}}\ \vec{x}; c$) **do** P "read" the constraint c. Furthermore, **whenever** c **do** Q stands for ($\underline{\textbf{wait}}\ \vec{x}; c$) **do** Q when $|\vec{x}| = 0$, i.e., $\vec{x} = \varepsilon$ (see Notation 6.5.1).

Let us briefly provide intuitions on this encoding. Consider the HVK processes

$$
\begin{array}{lcl}
P & = & \textbf{request}\ a(k)\ \textbf{in}\ P' \\
Q & = & \textbf{accept}\ a(x)\ \textbf{in}\ Q'
\end{array}
$$

The encoding of P declares a new variable session k and sends it through the channel a by posting the constraint $\mathtt{req}(a, k)$. Once $\mathsf{H}[\![Q]\!]$ receives the session key (local variable) generated by $\mathsf{H}[\![P]\!]$, it adds the constraint $\mathtt{acc}(a, k)$ to notify the acceptance of k. Then, $\mathsf{H}[\![P]\!]$ and $\mathsf{H}[\![Q]\!]$ synchronize using this constraint and they execute their continuation in the next time unit. Label selection and branching synchronize on the constraint $\mathtt{sel}(k, l)$. We use the parallel composition $\prod_{1 \le i \le n}$ **when** $l = l_i$ **do next** $\mathsf{H}[\![P_i]\!]$ to execute the selected choice. Notice that we do not require a non-deterministic choice since the constraints $l = l_i$ are mutually exclusive [Falaschi 1997, Nielsen 2002a].

As in [Honda 1998], in the encoding of **if** e **then** P **else** Q , we assume an evaluation function on expressions. Once e is evaluated, $\downarrow e$ is a *constant* boolean value.

The encoding of **def** D **in** P exploits the scheme described in Section 3.3.1 to define recursive definitions in utcc making use of abstractions.

Note that the encoding above delays the execution of the continuation of a process (e.g. P' in **request** $a(k)$ **in** P') to the next time unit (i.e., **next** $\mathsf{H}[\![P']\!]$). Therefore, we shall establish the correspondence between the HVK transition $\longrightarrow_{\mathtt{HVK}}$ and the observable transition $\Longrightarrow$ as we explain in the next section.

$$
\begin{array}{rcl}
\mathsf{H}[\![\textbf{request } a(k) \textbf{ in } P]\!] & = & (\textbf{local } k)\,(\underline{\textbf{tell}}(\texttt{req}(a,k)) \parallel \textbf{whenever } \texttt{acc}(a,k) \textbf{ do next } \mathsf{H}[\![P]\!]) \\
\mathsf{H}[\![\textbf{accept } a(k) \textbf{ in } P]\!] & = & (\underline{\textbf{wait}}\ k; \texttt{req}(a,k))\ \textbf{do}\ (\textbf{tell}(\texttt{acc}(a,k)) \parallel \textbf{next}\ (\mathsf{H}[\![P]\!])) \\
 & & \\
\mathsf{H}[\![k![\vec{e}]; P]\!] & = & \underline{\textbf{tell}}(\texttt{out}(k,\vec{e})) \parallel \textbf{whenever } \overline{\texttt{out}(k,\vec{e})} \textbf{ do next } \mathsf{H}[\![P]\!] \\
\mathsf{H}[\![k?(\vec{x}) \textbf{ in } P]\!] & = & (\underline{\textbf{wait}}\ \vec{x}; \texttt{out}(k,\vec{x}))\ \textbf{do next}\ \mathsf{H}[\![P]\!] \\
 & & \\
\mathsf{H}[\![k \lhd l; P]\!] & = & \underline{\textbf{tell}}(\texttt{sel}(k,l)) \parallel \textbf{whenever } \overline{\texttt{sel}(k,l)} \textbf{ do next } \mathsf{H}[\![P]\!] \\
\mathsf{H}[\![k \rhd \{l_1 : P_1 || \ldots || l_n : P_n\}]\!] & = & (\underline{\textbf{wait}}\ l; \texttt{sel}(k,l))\ \textbf{do} \prod\limits_{1 \le i \le n} \textbf{when } l = l_i \textbf{ do next } \mathsf{H}[\![P_i]\!] \\
\mathsf{H}[\![\textbf{throw } k[k']; P]\!] & = & \underline{\textbf{tell}}(\texttt{outk}(k,k')) \parallel \textbf{whenever } \overline{\texttt{outk}(k,k')} \textbf{ do next } \mathsf{H}[\![P]\!] \\
\mathsf{H}[\![\textbf{catch } k(k') \textbf{ in } P]\!] & = & (\underline{\textbf{whenever}}\ \texttt{outk}(k,k'))\ \textbf{do next}\ \mathsf{H}[\![P]\!] \\
 & & \\
\mathsf{H}[\![\textbf{if } e \textbf{ then } P \textbf{ else } Q\ ]\!] & = & \textbf{when } e \downarrow \texttt{true}\ \textbf{ do next } \mathsf{H}[\![P]\!] \parallel \textbf{when } e \downarrow \texttt{false}\ \textbf{ do next } \mathsf{H}[\![Q]\!] \\
\mathsf{H}[\![P \mid Q]\!] & = & \mathsf{H}[\![P]\!] \parallel \mathsf{H}[\![Q]\!] \\
\mathsf{H}[\![\textbf{inact}]\!] & = & \textbf{skip} \\
\mathsf{H}[\![(\nu u)P]\!] & = & (\textbf{local } u)\,\mathsf{H}[\![P]\!] \\
\mathsf{H}[\![\textbf{def } D \textbf{ in } P]\!] & = & \prod\limits_{X_i(x_i k_i) \in D} (\ulcorner X_i(x_i k_i) \urcorner \parallel \widehat{\mathsf{H}[\![P]\!]})
\end{array}
$$

Table 9.4: An Encoding from HVK into utcc. $\ulcorner \cdot \urcorner$ and $\widehat{P}$ as in Definition 3.3.2.

9.3.1 Operational Correspondence

In this section we prove the correctness of our encoding. For the sake of simplicity, without loss of generality, we assume programs of the form **def** D **in** P where there are not procedure definitions in P.

Let us introduce the following normal form of HVK processes.

Definition 9.3.1 (Processes in normal form)**.** *We say that the HVK process P is in* normal form *if takes the form* **def** D **in** $\nu\vec{u}(Q_1 \mid \cdots \mid Q_n)$ *where neither the operators "ν" and "$\mid$" nor process variables occur in the top level of $Q_1, \ldots, Q_n$.*

The following proposition states that given a process P we can find P' in normal form such that: either P' is structurally congruent to P or results from replacing the process variables in the top level of P by their corresponding definition using the rule Def .

Proposition 9.3.1. *For all HVK process P there exists P' in normal form s.t. $P \longrightarrow^*_{HVK} \equiv P'$ only using the rules* Def *and* Str *in Table 9.2.*

Proof. Let P be a process of the form **def** D **in** Q where there are no procedure definitions in Q. By repeated applications of the rule Def, we can show that $P \longrightarrow^*_{\texttt{HVK}} P'$ where P' does not have occurrences of processes variables in the top level. Then, we use the rules of the structural congruence to move the local variables to the outermost position and find $P'' \equiv P'$ in the desired normal form. □

Notice that the rules of the operational semantics of HVK are given for pairs of processes that can interact with each other. We shall refer to those pairs of processes as *redex*.

Definition 9.3.2 (Redex)**.** *A* redex *is a pair of dual processes composed in parallel as in*

- **request** $a(k)$ **in** P $\mid$ **accept** $a(k)$ **in** Q
- $k![\vec{e}]; P$ $\mid$ $k?(\vec{x})$ **in** Q
- $k \lhd l; P$ $\mid$ $k \rhd \{l_1 : P_1 \parallel \cdots || l_n : P_n\}$

- **throw** $k[k']; P \;\mid$ **catch** $k(k')$ **in** Q.

It is worth noticing that a redex in HVK synchronizes and reduces in a single transition as in $(k![\vec{e}]; P) \mid (k?(x)\,\mathbf{in}\,Q) \longrightarrow_{\texttt{HVK}} P \mid Q[\vec{e}/\vec{x}]$. Nevertheless, in utcc, the encoding of the processes above requires two internal transitions: one for adding the constraint $\texttt{out}(k, \vec{e})$ to the current store, and another one in which the process $(\underline{\textbf{wait}}\ \vec{x}; \texttt{out}(k, \vec{x}))$ **do next** $[\![Q]\!]$ "reads" that constraint to later execute **next** $[\![Q[\vec{e}/\vec{x}]]\!]$. We shall then establish the operational correspondence between an observable transition of utcc (obtained from a finite number of internal transitions) and the following reduction relation over HVK processes:

Definition 9.3.3 (Outermost Reductions)**.** *Let $P \equiv$ **def** D **in** $\nu\vec{x}(Q_1 \mid \cdots \mid Q_n)$ be an HVK program in normal form. We define the outermost reduction relation $P \Longrightarrow_{\texttt{HVK}} P'$ as the maximal sequence of reductions $P \longrightarrow^*_{\texttt{HVK}} P' \equiv$ **def** D **in** $\nu\vec{x'}(Q'_1 \mid \cdots \mid Q'_n)$ such that for every $i \in \{1, ..n\}$, either*

1. *$Q_i =$ **if** e **then** R_1 **else** $R_2 \longrightarrow_{\texttt{HVK}} R_{1/2} = Q'_i$;*
2. *for some $j \in \{1, ..n\}$, $Q_i|Q_j$ is a redex such that $Q_i|Q_j \longrightarrow_{\texttt{HVK}} \nu\vec{y}(Q'_i|Q'_j)$, with $\vec{y} \subseteq \vec{x'}$;*
3. *there is no $k \in \{1, ..n\}$ such that $Q_i \mid Q_k$ is a redex and $Q_i \equiv Q'_i$.*

In addition to the difference between the synchronous and the asynchronous nature of HVK and utcc, there is another fundamental difference between both languages that we need to consider to establish the semantic correspondence. Namely, utcc is a deterministic languages while HVK may exhibit non-deterministic behavior. In the following we explain why this difference is not relevant when considered *well-typed* HVK processes.

Observation 9.3.1 (Typable HVK processes)**.** *In the π-calculus, and in HVK, inputs and outputs are not necessarily persistent. Then, the parallel composition of two outputs and one input on the same channel may lead to different configurations. Take for example $P = k![\vec{e}]; Q_1 \mid k![\vec{e'}]; Q_2 \mid k?(\vec{x})\,\mathbf{in}\,Q_3$. We have one of the following derivations:*

- $P \longrightarrow_{\texttt{HVK}} Q_1 \mid k![\vec{e'}]; Q_2 \mid \ Q_3[\vec{e}/\vec{x}]$
- $P \longrightarrow_{\texttt{HVK}} Q_2 \mid k![\vec{e}]; Q_1 \mid \ Q_3[\vec{e'}/\vec{x}]$

In both cases, there is an output that cannot interact with the input $k?(\vec{x})\,\mathbf{in}\,Q_3$.

In utcc, inputs are represented by abstractions which are persistent during a time unit. Then, in the example above, we shall observe that both outputs react with the input, i.e., we observe that $\mathsf{H}[\![P]\!] \overset{(\texttt{true},c)}{\Longrightarrow} \mathsf{H}[\![Q_3[\vec{e}/\vec{x}]]\!] \parallel \mathsf{H}[\![Q_3[\vec{e'}/\vec{x}]]\!]$.

A similar situation arises when one considers a parallel composition of the form $P_1 \mid ... \mid P_n$ where there exist a process P_i that form a redex with two different processes P_j and P_k.

Here, to establish the semantic correspondence we appeal to the typed nature of the HVK language. Roughly speaking, the type discipline in [Honda 1998] ensures a correct "pairing" between complementary components, i.e., redex. Our encoding assumes then processes to be typable with respect to such a discipline.

In the sequel we shall thus consider only HVK processes P where for $n \geq 1$, if $P \equiv_h P_1 \Longrightarrow_h P_2 \Longrightarrow_h \cdots \Longrightarrow_h P_n$ and $P \equiv_h P'_1 \Longrightarrow_h P'_2 \Longrightarrow_h \cdots \Longrightarrow_h P'_n$ then $P_i \equiv_h P'_i$ for all $i \in \{1, .., n\}$, i.e., P is a *deterministic* process.

Recall the Notation 6.3.1 for the utcc internal and observable transitions $P \longrightarrow Q$ and $P \Longrightarrow Q$ respectively where the inputs are assumed to be true and the outputs unimportant. We shall also use the observable equivalence relation $\sim^{obs}$ in Definition 6.5.3 that ignores the residual processes generated by the evolution of the processes of the form $\underline{\textbf{tell}}(c(\vec{t})) \parallel (\underline{\textbf{wait}}\ \vec{x}; c)$ **do** Q (see Notation 6.5.2).

Theorem 9.3.1 (Operational Correspondence)**.** *Let P, Q be typable HVK processes in normal form and R,S be utcc processes. It holds:*
1) **Soundness***: If $P \Longrightarrow_{HVK} Q$, then there exists R s.t. $\mathsf{H}[\![P]\!] \Longrightarrow R \sim^{obs} \mathsf{H}[\![Q]\!]$*
2) **Completeness***: If $\mathsf{H}[\![P]\!] \Longrightarrow S$, then there exists Q s.t. $P \Longrightarrow_{HVK} Q$ and $\mathsf{H}[\![Q]\!] \sim^{obs} S$.*

Proof. Assume that

$$\begin{array}{lll} P & \equiv_h & \textbf{def } D \textbf{ in } \nu\vec{x}(Q_1 \mid \cdots \mid Q_n) \\ Q & \equiv_h & \textbf{def } D \textbf{ in } \nu\vec{x'}(Q'_1 \mid \cdots \mid Q'_n) \end{array}$$

1. *Soundness.* Since $P \Longrightarrow_h Q$ there must exist a sequence of derivations of the form $P \equiv_h P_1 \longrightarrow_{\mathtt{HVK}} P_2 \longrightarrow_{\mathtt{HVK}} \ldots \longrightarrow_{\mathtt{HVK}} P_n \equiv_h Q$. The proof proceeds by induction on the length of this derivation, with a case analysis on the last applied rule. We then have the following cases:

   (a) **Using the rule IF1**. It must be the case that there exists $Q_i \equiv_h \textbf{if } e \textbf{ then } R_1 \textbf{ else } R_2$ and $Q_i \longrightarrow_{\mathtt{HVK}} R_1 \equiv_h Q'_i$ and $e \downarrow \mathtt{true}$. One can easily show that $\textbf{when } e \downarrow \mathtt{true} \textbf{ do next } [\![Q'_i]\!] \Longrightarrow [\![Q'_i]\!]$.

   (b) **Using the rule IF2** Similarly as for IF1.

   (c) **Using the rule LINK**. It must be the case that there exist i, j such that $Q_i \equiv_h \textbf{request } a(k) \textbf{ in } Q'_i$ and $Q_j \equiv_h \textbf{accept } a(x) \textbf{ in } Q'_j$ and then $Q_i \mid Q_j \longrightarrow_{\mathtt{HVK}} (\nu k)(Q'_i \mid Q'_j)$. We then have a derivation of the form

$$\begin{array}{rll} [\![Q_i]\!] \parallel [\![Q_k]\!] & \longrightarrow^* & (\textbf{local } k; c)\,(R'_i \parallel \textbf{whenever } \mathtt{acc}(a,k) \textbf{ do next } [\![Q'_i]\!] \parallel \\ & & \qquad (\underline{\textbf{wait}}\ k'; \mathtt{req}(a,k')) \textbf{ do } (\textbf{tell}(\mathtt{acc}(a,k')) \parallel \\ & & \qquad\qquad \textbf{next } ([\![Q'_j]\!])) \\ & \longrightarrow^* & (\textbf{local } k; c')\,(R'_i \parallel \textbf{whenever } \mathtt{acc}(a,k) \textbf{ do next } [\![Q'_i]\!] \parallel \\ & & \qquad R'_j \parallel \textbf{tell}(\mathtt{acc}(a,k)) \parallel \textbf{next } ([\![Q'_j[k/k']]\!]) \\ & \longrightarrow^* & (\textbf{local } k; c'')\,(R'_i \parallel R'_j \parallel \textbf{next } [\![Q'_i]\!] \parallel \textbf{next } ([\![Q'_j[k/k']]\!]) \not\longrightarrow \end{array}$$

   where $c = \mathtt{req}(a,k)$, $c' = c \wedge \overline{\mathtt{req}(a,k)}$, $c'' = c' \wedge \mathtt{acc}(a,k) \wedge \overline{\mathtt{acc}(a,k)}$ and R'_i, R'_j are the processes resulting after the interaction of the processes in the parallel composition $\underline{\textbf{tell}}(\mathtt{req}(a,k)) \parallel (\underline{\textbf{wait}}\ k'; \mathtt{req}(a,k'))$ **do** $\cdots$, i.e.:

$$\begin{array}{rll} R'_i & \equiv_u & (\textbf{local } go, stop; \mathtt{out}'(go) \wedge \mathtt{out}'(stop) \wedge c(\vec{t})) \\ & & \qquad \textbf{next}\,!\,\textbf{unless } \mathtt{out}'(stop) \textbf{ next tell}(\mathtt{out}'(go)) \parallel \\ & & \qquad \textbf{next}\,!\,\textbf{tell}(\mathtt{out}'(stop)) \\ R'_j & \equiv_u & (\textbf{local } stop', go'; \mathtt{out}'(go') \wedge \overline{c}(\vec{t}) \wedge \mathtt{out}'(stop'))\,\textbf{next}\,!\,\textbf{tell}(\mathtt{out}'(stop')) \\ & & \quad \parallel \textbf{next}\,!\,\textbf{unless } \mathtt{out}'(stop') \textbf{ next tell}(\mathtt{out}'(go')) \\ & & \quad \parallel (\textbf{abs } \vec{x}; c \wedge \mathtt{out}'(go') \wedge \vec{x} \neq \vec{t})\,(Q \parallel \textbf{tell}(\overline{c}(\vec{t})) \parallel !\,\textbf{tell}(\mathtt{out}'(stop')) \\ & & \quad \parallel \textbf{next}\,!\,(\textbf{abs } \vec{x}; c \wedge \mathtt{out}'(go'))\,(Q \parallel \textbf{tell}(\overline{c}(\vec{t})) \parallel !\,\textbf{tell}(\mathtt{out}'(stop')) \end{array}$$

   We notice that $R'_i \parallel R'_j \not\longrightarrow$ and it is a process that can only output the constraint $\mathtt{out}'(x)$ where x is a local variable. By appealing to Observation 6.5.1 we conclude $[\![Q_i]\!] \parallel [\![Q_j]\!] \Longrightarrow \sim^{obs} (\textbf{local } k)\,([\![Q'_i]\!] \parallel [\![Q'_j]\!])$.

   (d) The cases using the rules LABEL and PASS can be proven similarly as the case for the rule LINK.

2. *Completeness.* Given the encoding and the structure of P, we have a utccprocess $R = [\![P]\!]$ such that

$$R \equiv_u (\textbf{local } \vec{x})\,([\![Q_1]\!] \parallel \ldots \parallel [\![Q_n]\!])\,.$$

Let $R_i = [\![Q_i]\!]$ for $1 \leq i \leq n$. By an analysis on the structure of R, if $R_i \longrightarrow R'_i$ then it must be the case that either (a) $R_i = \mathbf{when}\ e\ \mathbf{do\ next}\ [\![Q'_i]\!]$ and $R'_i = \mathbf{next}\ [\![Q'_i]\!]$ or (b) $\langle R_i, c\rangle \longrightarrow \langle R'_i, c \wedge d\rangle$ where d is a constraint of the form $\mathtt{req}(\cdot)$, $\mathtt{sel}(\cdot)$, $\mathtt{out}(\cdot)$, or $\mathtt{outk}(\cdot)$. In both cases we shall show that there exists a R''_i such that $R_i \longrightarrow^* R''_i \not\longrightarrow$ such that $Q_i \longrightarrow_{\mathtt{HVK}} Q'_i$ and $R''_i = \mathbf{next}\,[\![Q'_i]\!]$.

(a) Assume that $R_i = \mathbf{when}\ e \downarrow \mathtt{true}\ \mathbf{do\ next}\,[\![Q'_i]\!]$ for some Q'_i. Then it must be the case that $Q_i = \mathbf{if}\,e\ \mathbf{then}\ Q'_i\ \mathbf{else}\ Q''_i$. If $e \downarrow \mathtt{true}$ we then have $R''_i = \mathbf{next}\,[\![Q'_i]\!]$. The case when $e \downarrow \mathtt{false}$ is similar by considering $R_i = \mathbf{when}\ e \downarrow \mathtt{false}\ \mathbf{do}\ Q'_i$.

(b) Assume now that $\langle R_i, c\rangle \longrightarrow \langle R'_i, c \wedge d\rangle$ where d is of the form $\mathtt{req}(\cdot)$, $\mathtt{sel}(\cdot)$, $\mathtt{out}(\cdot)$ or $\mathtt{outk}(\cdot)$. We proceed by case analysis of the constraint d. Let us consider only the case $d = \exists_k(\mathtt{req}(a,k))$; the cases in which d takes the form $\mathtt{sel}(\cdot)$, $\mathtt{out}(\cdot)$, or $\mathtt{outk}(\cdot)$ are handled similarly. If $d = \exists_k(\mathtt{req}(a,k))$ for some a, then we must have that $Q_i \equiv_h \mathbf{request}\ a(k)\ \mathbf{in}\ Q'_i$ for some i. If there exists j such that $Q_j \equiv_h \mathbf{accept}\ a(x)\ \mathbf{in}\ Q'_j$, one can show a derivation similar to the case of the rule Link in soundness to prove that $R_i \parallel R_j \longrightarrow^* \sim^o (\mathbf{local}\,k)\,(\mathbf{next}\,[\![Q'_i]\!] \parallel \mathbf{next}\,[\![Q'_j]\!])$. If there is no Q_j such that $Q_i \,|Q_j$ forms a redex, then one can show that $R_i \Longrightarrow \sim^{obs} R_i$.

□

9.4 HVK-T: An Temporal extension of HVK

In this section we propose HVK-T, a temporal extension of HVK in which a bundled treatment of time is explicit and session closure is considered. More precisely, the HVK-T language arises as the extension of HVK processes with refined constructs for session request and acceptance, as well as with a construct for session abortion.

Definition 9.4.1 (HVK-T syntax)**.** *HVK-T processes are given by the following syntax:*

P	::=	$\mathbf{request}\ a(k)\ \mathbf{during}\ m\ \mathbf{in}\ P$	*Timed Session Request*
	|	$\mathbf{accept}\ a(k)\ \mathbf{given}\ c\ \mathbf{in}\ P$	*Declarative Session Acceptance*
	|	$\cdots$	*{ the other constructs, as in Def. 9.2.1 }*
	|	$\mathbf{kill}\,c_k$	*Session Abortion*

The intuition behind these three operators is the following: $\mathbf{request}\ a(k)\ \mathbf{during}\ m\ \mathbf{in}\ P$ will request a session k over the service name a during m time units. Its dual construct is $\mathbf{accept}\ a(k)\ \mathbf{given}\ c\ \mathbf{in}\ P$: it will grant the session key k when requested over the service name a provided by a session and a successful check over the constraint c. Notice that c stands for a precondition for agreement between session request and acceptance. In c, the duration m of the corresponding session key k can be referenced by means of the variable dur_k. In the encoding we syntactically replace it by the variable corresponding to m. Finally, $\mathbf{kill}\,c_k$ will remove c_k from the valid set of sessions.

Adapting the encoding in Table 9.4 to consider HVK-T processes is remarkably simple. Indeed, modifications to the encoding of session request and acceptance are straightforward. The most evident change is the addition of the parameter m within the constraint $\mathtt{req}(a,k,m)$. The duration of the requested session is suitably represented as a bounded replication ($!_{[m]}$) of the process defining the activation of the session k represented as the constraint $\mathtt{act}(k)$. The execution of the continuation $\mathsf{H}[\![P]\!]$ is guarded by the constraint $\mathtt{act}(k)$ (i.e., P can be executed only when the session k is valid). In the encoding, we use

$$
\begin{array}{rcl}
\mathsf{H}[\![\textbf{request } a(k) \textbf{ during } m \textbf{ in } P]\!] & = & (\textbf{local}\, k)\, \underline{\textbf{tell}}(\texttt{req}(a,k,m)) \parallel \\
& & \quad \textbf{whenever } \texttt{acc}(a,k) \textbf{ do next} (\\
& & \quad\quad \textbf{tell}(\texttt{act}(k)) \parallel \mathcal{G}_{\texttt{act}(k)}(\mathsf{H}[\![P]\!]) \parallel \\
& & \quad\quad !_{[m]} \textbf{unless } \texttt{kill}(k) \textbf{ next tell}(\texttt{act}(k))) \\
\mathsf{H}[\![\textbf{accept } a(k) \textbf{ given } c \textbf{ in } P]\!] & = & (\underline{\textbf{wait}}\ k; \texttt{req}(a,k,m) \wedge c[m/dur_k])\ \textbf{do}\ (\\
& & \quad \textbf{tell}(\texttt{acc}(a,k)) \parallel \textbf{next}\, \mathcal{G}_{\texttt{act}(k)}(\mathsf{H}[\![P]\!])) \\
\mathsf{H}[\![\textbf{kill}\, k]\!] & = & !\,\textbf{tell}(\texttt{kill}(k))
\end{array}
$$

Table 9.5: The Extended Encoding. $\mathcal{G}_d(P)$ is given in Definition 9.4.2. The process $!_{[m]}P$ means $P \parallel \textbf{next}\, P \parallel ... \parallel \textbf{next}^{\,m} P$.

the function $\mathcal{G}_d(P)$ to stand for the process which behaves as P when the constraint d can be entailed from the current store, and that is precluded from execution otherwise. More precisely,

Definition 9.4.2. *Let $\mathcal{G}_d :\rightarrow Procs \rightarrow Procs$ be defined as*

$$
\mathcal{G}_d(P) = \begin{cases}
\textbf{skip} & \text{if } P = \textbf{skip} \\
\textbf{when } d \textbf{ do tell}(c) & \text{if } P = \textbf{tell}(c) \\
(\textbf{abs } \vec{x}; c)\, \mathcal{G}_d(Q) & \text{if } P = (\textbf{abs } \vec{x}; c)\, Q \text{ and } \vec{x} \notin fv(d) \\
\mathcal{G}_d(P_1) \parallel \mathcal{G}_d(P_2) & \text{if } P = P_1 \parallel P_2 \\
(\textbf{local}\, \vec{x}; c)\, \mathcal{G}_d(Q) & \text{if } P = (\textbf{local}\, \vec{x}; c)\, Q \text{ and } \vec{x} \notin fv(d) \\
\textbf{when } d \textbf{ do next}\, \mathcal{G}_d(Q) & \text{if } P = \textbf{next}\, Q \\
\textbf{when } d \textbf{ do unless } c \textbf{ next}\, \mathcal{G}_d(Q) & \text{if } P = \textbf{unless } c \textbf{ next}\, Q \\
!\,\mathcal{G}_d(Q) & \text{if } P = !\,Q
\end{cases}
$$

In the side of session acceptance, the main novelty is the introduction of $c[m/dur_k]$. As explained before, we syntactically replace the variable dur_k by the corresponding duration of the session m. This is a generic way to represent the agreement that should exist between a service provider and a client; for instance, it could be the case that the client is requesting a session longer than what the service provider can or want to grant.

9.4.1 Case Study: Electronic booking

Here we present an example in a electronic booking scenario that makes use of the constructs introduced in `HVK-T`.

Consider the electronic portal of a airline company AC which offers flights online. A customer makes us of this service by establishing a timed session with AC. The costumer may ask for a flight proposal given a set of constraints such as dates allowed, destinations, etc. After receiving an offer from AC, the customer checks, for example, that the price is below a given threshold. If so, she accepts the proposal initializing the contract phase. One possible `HVK-T` specification of this scenario is described in Table 9.6.

In the specification of the service AC, the process checks if the duration of the session requested by the customer is less than a parameter `maxtime`. This service also closes the session if the customer rejects the proposal. This way, dangling sessions are avoided.

Customer	=	**request** $ob(k)$ **during** m **in** $(k![bookingdata]; Select(k))$
Select(k)	=	$k?(offer)$ **in** (**if** $(offer.price \leq 1500)$ **then** $k \triangleleft Contract$; **else** $k \triangleleft Reject$;)
AC	=	**accept** $ob(k)$ **given** $dur_m \leq$ `maxtime` **in** (
		$k?(bookingData)$ **in**
		$(\nu u)k![u]; k \rhd \{Contract : \overline{Accept} \mid Reject : \mathbf{kill}\, k\})$

Table 9.6: Online booking example

9.4.2 Exploiting the Logic Correspondence

We can draw inspiration from the *constraint templates* put forward in [Pesic 2006], a set of LTL formulas that represent desirable/undesirable situations in service management. Such templates are divided in three types: *existence constraints*, that specify the number of executions of an activity; *relation constraints*, that define the relation between two activities to be present in the system; and *negation constraints*, which are essentially the negated versions of relation constraints. Appealing to the logic characterization of `utcc` processes as FLTL formulae, we may verify the existence and relation constraints over `HVK-T` programs. Assume a `HVK-T` program P and let $F = \mathsf{TL}[\![\mathsf{H}[\![P]\!]]\!]$ (i.e., the FLTL formula associated to the `utcc` representation of P). For existence constraints, assume that P defines a service accepting requests on channel a. If the service is eventually active, then it must be the case that $F \models \Diamond\exists_k(\mathtt{acc}(a,k))$ (recall that the encoding of **accept** adds the constraint $\mathtt{acc}(a,k)$ when the session k is accepted).

This way, additional to the behavior techniques, `utcc` may offer also declarative reasoning techniques based upon ratability in FLTL.

⋆ ⋆ ⋆

9.5 Multimedia Interaction Systems

Interactivity in multimedia systems has become increasingly important. The aim is to devise ways for the machine to be an effective and active partner in a collective behavior constructed dynamically by many actors. In its simplest form, a person (say a musician) signals the computer when specific previously defined processes should be launched or stopped. In more complex forms of interaction the machine is always actively adapting its behavior according to the information derived from the activity of the other partners who, in turn, adapt theirs according to the computer actions. To be coherent these machine actions must be the result of a complex adaptive system, composed itself of many agents that should be coordinated in precise ways. Constructing such systems is thus a challenging task. Moreover, ensuring their correctness poses a great burden to the usual test-based techniques. In this setting, CCP-based languages has much to offer: CCP calculi are explicitly designed for expressing complex coordination patterns in a very simple way by means of constraints. In addition, their declarative nature allows formally proving properties of interactive systems modeled with them.

Interactive scores [Allombert 2007] can be seen as models for reactive music systems adapting their behavior to different types of intervention from a performer. The weakly defined temporal relations between the components in an interactive score specify loosely coupled music processes potentially changing their properties (temporal, harmonic, etc.) in reaction to stimulus from the environment (a performer, another machine, etc). An

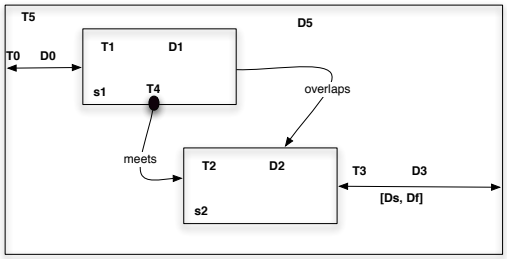


Figure 9.3: Interactive score

interactive score defines a hierarchical structure of processes. Musical properties of a process depend on the hierarchical context it is located in. Although the hierarchical structure has been treated as static in previous works on interactive scores [Allombert 2007], there is no reason it should be so. A process, in reaction to a musician action, for example, could be programmed to move from one context to another or simply to disappear. One can imagine, for instance, a particular set of musical materials within different contexts that should only be played when an expected information from the environment actually takes place. Modeling this kind of interactive score mobility in a coherent way is greatly simplified by using `utcc`.

Musical improvisation is another natural context for interacting agents. Improvisation is effective when the behavior of the agents adapts to what has been learned in previous interactions. A music style-learning/improvisation scheme such as Factor Oracle [Allauzen C. 1999, Assayag 2006] can also be seen as a reactive system where several learning and improvising agents react to information provided by the environment or by other agents. In its simplest form three concurrent agents, a player, a learner and an improviser must be synchronized. Since only three independent processes are active, coordination can be implemented without major difficulties using traditional languages and tools. The question is whether such implementations would scale up to situations involving several players, learners and improvisers. For an implementation using traditional languages the complexity of such systems would most likely impose many simplifications in coordination patterns if behavior is to be controlled in a significant way. A `utcc` model, as described here, provides a very compact and simple model of the agents involved in the FO improvisation, one in which coordination is automatically provided by the blocking *ask* construct of the calculus. Moreover, additional agents could easily be incorporated in the system. As an extra bonus, fundamental properties of the constructed system can be formally verified in the model.

Here we argue for `utcc` as a declarative language for the modeling and verification of dynamic multimedia interaction systems. We shall show that its extra expressiveness to model mobile behavior allow us to neatly define more flexible and dynamic systems. More precisely, we shall present a `utcc` model for interactive scores where the interactive points allow the composer to dynamically change the hierarchical structure of the score. We then broaden the interaction mechanisms available for the user in previous (more static) models, e.g., [Allombert 2006]. Furthermore, we propose a model for the Factor Oracle improvisation scheme that is simpler than that in [Assayag 2006].

9.6 Dynamic Interactive Scores

An interactive score [Allombert 2007] is a pair composed of *temporal objects* and Allen temporal relations [Allen 1983]. In general, each object is comprised of a start-time, a duration, and a procedure. The first two can be partially specified by constraints, with different constraints giving rise to different types of temporal objects, so-called *events* (duration equals zero), *textures* (duration within some range), *intervals* (textures without procedures) or *control-points* (a temporal point occurring somewhere within an interval object). The procedure gives operational meaning to the action of the temporal object. It could just be playing a note or a chord, or any other action meaningful for the composer. Figure 9.3, based on one from [Allombert 2007], shows an interactive score where temporal objects are represented as *boxes*. Objects are T_i, durations D_i. Object T_4 is a control point, whereas T_0 and T_3 are intervals. Duration D_3 should be such that $D_s \leq D_3 \leq D_f$.

The whole temporal structure is determined by the hierarchy of temporal objects. Suppose that, as a result of the information obtained by the occurrence of an event, object T_2 should no longer synchronize with a control-point inside T_1 but, say, with a similar point inside T_5. This very simple interaction cannot be modeled in the standard model of interactive scores [Allombert 2007]. Another example is an object waiting for some interaction from the performer within some temporal interval. If the interaction does not occur, the composer might then determine to probe the environment again later when a similar musical context has been defined. This amounts to moving the waiting interval from one box to another.

9.6.1 A `utcc` model for Dynamic Interactive Scores

Figure 9.4 shows our model for dynamic interactive scores. The process *BoxOperations* may perform the following actions:

- `mkbox`(id, d): defines a new box with id id and duration d. The start time is defined as a new (local) variable s whose value will be constrained by the other processes.
- `destroy`(id): firstly, it retrieves the box *sup* which contains the box id. If the box id is not currently playing, in the next time unit, it drops the boundaries of id by inserting all the boxes contained in id into *sup*.
- `before`(x, y): checks if x and y are contained in the same box. If so, the constraint `bf`(x, y) is added.
- `into`(x, y): dictates that the box x is into the box y if x is not currently playing.
- `out`(x, y): takes the box x out of the box y if x is not currently playing.

Process *Constraints* adds the necessary constraints relating the start times of each temporal object to respect the hierarchical structure of the score. For each constraint of the form `in`(x, y), this process dictates that the start time of x must be less than the one of y. Furthermore, the end time of y (i.e. $d_y + s_y$) must be greater than the end time of x. The case for `bf`(x, y) can be explained similarly.

The process *Persistence* transfers the information of the hierarchy (i.e. box declarations, `in` and `bf` relations) to the next time unit.

The process *Clock* defines a simple clock that binds the variable t to the value v in the current time unit and to $v + 1$ in the next time unit.

$$
\begin{array}{rcl}
\mathit{BoxOperations} & \stackrel{\text{def}}{=} & (\textbf{abs}\ id, d; \texttt{mkbox}(id, d)) \\
& & \qquad (\textbf{local}\ s)\ \textbf{tell}(\texttt{box}(id, d, s)) \\
& & \parallel (\textbf{abs}\ id; \texttt{destroy}(id)) \\
& & \qquad (\textbf{abs}\ x, sup; \texttt{in}(x, id) \wedge \texttt{in}(id, sup)) \\
& & \qquad\qquad \textbf{unless}\ \texttt{play}(id)\ \textbf{next}\ \textbf{tell}(\texttt{in}(x, sup)) \\
& & \parallel (\textbf{abs}\ x, y; \texttt{before}(x, y))\ \textbf{when}\ \exists_z(\texttt{in}(x, z) \wedge \texttt{in}(y, z))\ \textbf{do} \\
& & \qquad\qquad \textbf{unless}\ \texttt{play}\,(y)\ \textbf{next}\ \textbf{tell}(\texttt{bf}(x, y)) \\
& & \parallel (\textbf{abs}\ x, y; \texttt{into}(x, y))\ \textbf{unless}\ \texttt{play}\,(x)\ \textbf{next}\ \textbf{tell}(\texttt{in}(x, y)) \\
& & \parallel (\textbf{abs}\ x, y; \texttt{out}(x, y))\ \textbf{when}\ \texttt{in}(x, y)\ \textbf{do} \\
& & \qquad\qquad \textbf{unless}\ \texttt{play}\,(x)\ \textbf{next}\ (\textbf{abs}\ z, \texttt{in}(y, z); \textbf{tell}(\texttt{in}(x, z))) \\
\mathit{Constraints} & \stackrel{\text{def}}{=} & (\textbf{abs}\ x, y; \texttt{in}(x, y))\ (\textbf{abs}\ d_x, s_x; \texttt{box}(x, d_x, s_x)) \\
& & \qquad (\textbf{abs}\ d_y, s_y; \texttt{box}(y, d_y, s_y)) \\
& & \qquad\quad \textbf{tell}(s_y \leq s_x) \parallel \textbf{tell}(d_x + s_x \leq d_y + s_y) \\
& & \parallel (\textbf{abs}\ x, y; \texttt{bf}(x, y))\ (\textbf{abs}\ d_x, s_x; \texttt{box}(x, d_x, s_x)) \\
& & \qquad\qquad (\textbf{abs}\ d_y, s_y; \texttt{box}(y, d_y, s_y))\ \textbf{tell}(s_x + d_x \leq s_y) \\
\mathit{Persistence} & \stackrel{\text{def}}{=} & (\textbf{abs}\ x, y; \texttt{in}(x, y))\ \textbf{when}\ \texttt{play}(x)\ \textbf{do}\ \textbf{next}\ \textbf{tell}(\texttt{in}(x, y)) \\
& & \qquad \parallel \textbf{unless}\ \texttt{out}(x, y) \vee \texttt{destroy}(x)\ \textbf{next}\ \textbf{tell}(\texttt{in}(x, y)) \\
& & \parallel (\textbf{abs}\ x, y; \texttt{bf}(x, y))\ \textbf{when}\ \texttt{play}(y)\ \textbf{do}\ \textbf{next}\ \textbf{tell}(\texttt{bf}(x, y)) \\
& & \qquad\qquad \parallel \textbf{unless}\ (\texttt{out}(x, y) \vee \texttt{destroy}(y)\ \textbf{next}\ \textbf{tell}(\texttt{bf}(x, y)) \\
& & \parallel (\textbf{abs}\ x; \texttt{box}(x, d_x, s_x))\ \textbf{when}\ \texttt{play}(x)\ \textbf{do}\ \textbf{next}\ \textbf{tell}(\texttt{box}(x, d_x, s_x)) \\
& & \qquad\qquad\qquad \parallel \textbf{unless}\ \texttt{destroy}(x)\ \textbf{next}\ \textbf{tell}(\texttt{box}(x, d_x, s_x)) \\
\mathit{Clock}(t, v) & \stackrel{\text{def}}{=} & \textbf{tell}(t = v) \parallel \textbf{next}\, \mathit{Clock}(t, v + 1) \\
\mathit{Play}(x, t) & \stackrel{\text{def}}{=} & \textbf{when}\ t \geq 1\ \textbf{do}\ \textbf{tell}(\texttt{play}(x)) \parallel \textbf{unless}\ t \leq 1\ \textbf{next}\ \mathit{Play}(x, t - 1) \\
\mathit{Init}(t) & \stackrel{\text{def}}{=} & (\textbf{wait}\ x; \mathit{init}(x))\ \textbf{do} \\
& & \qquad (\textbf{abs}\ d_x, s_x; \texttt{box}(x, d_x, s_x)) \\
& & \qquad\qquad \mathit{Clock}(t, 0) \parallel \textbf{tell}(s_x = t) \parallel \\
& & \qquad\qquad !\,(\textbf{wait}\ y, d_y, s_y; \texttt{box}(y, d_y, s_y) \wedge s_y \leq t)\ \textbf{do}\ \mathit{Play}(y, d_y) \\
\\
\mathit{System} & \stackrel{\text{def}}{=} & (\textbf{local}\, t)\ \mathit{Init}(t) \parallel\!! \mathit{Persistence} \parallel\!! \mathit{Constraints} \parallel\!! \mathit{BoxOperations} \parallel \mathit{UsrBoxes}
\end{array}
$$

Figure 9.4: A `utcc` model for Dynamic Interactive Scores

The process $\mathit{Play}(x, t)$ adds the constraint $\texttt{play}(x)$ during t time units. This informs the environment that the box x is currently playing.

The process $\mathit{Init}(t)$ waits until the environment provides the constraint $\mathit{init}(x)$ for the outermost box x to start the execution of the system. Then, the *clock* is started and the start time of x is set to 0. The rest of the boxes wait until their start time is less or equal to the current time (t) to start playing.

Finally, the whole system is the parallel composition between the previously defined processes and the specific user model, for instance, the process $\mathit{UsrBoxes}$ in Figure 9.5.

This system defines the hierarchy in Figure 9.6(a). When b starts playing, the system asks if the signal `signal` is present (i.e., if it was provided by the environment). If it was not, the box d is taking out from the context b. Furthermore, a new box f is created such that b must be played before f and f before d as in Figure 9.6(b). Notice that when the box d is taken out from b, the internal box e is still into d preserving its structure.

9.6.2 Verification of the Model

The processes defined by the user may lead to situations where the final store is inconsistent as in $st < 5 \wedge st > 7$ where st is the start time of a given box. Take for example the process $\mathit{UsrBoxes}$ above. If the box f is defined with a duration greater than 5, the execution of f (and then that of d) will exceed the boundaries of the box a which contains both structures.

In this context, the declarative view of `utcc` processes as FLTL formulae provides a valuable tool for the verification of the model: The formula $A = \mathsf{TL}[\![P]\!]$ may allow us to verify whether the execution of P leads to an inconsistent store. Thus, we can detect pitfalls

$$
\begin{aligned}
UsrBoxes \stackrel{\text{def}}{=}\ & \textbf{tell}(\texttt{mkbox}(a,22) \wedge \texttt{mkbox}(b,12) \wedge \texttt{mkbox}(c,4)) \parallel \\
& \textbf{tell}(\texttt{mkbox}(d,5) \wedge \texttt{mkbox}(e,2)) \parallel \\
& \textbf{tell}(\texttt{into}(b,a) \wedge \texttt{into}(c,b) \wedge \texttt{into}(d,b) \wedge \texttt{into}(e,d)) \parallel \\
& \textbf{tell}(\texttt{before}(c,d)) \parallel \\
& \textbf{whenever } \texttt{play}(b) \textbf{ do unless } \texttt{signal next} \\
& \qquad\qquad \textbf{tell}(\texttt{out}(d,b) \wedge \texttt{mkbox}(f,2) \wedge \texttt{into}(f,a)) \parallel \\
& \qquad\qquad \textbf{tell}(\texttt{before}(b,f) \wedge \texttt{before}(f,d))
\end{aligned}
$$

Figure 9.5: Example of a Specification of Boxes

in the user model such as trying to place a bigger box into a smaller one or taking a box out of the outermost box.

In the following, we present two simple examples of temporal properties we could verify in an interactive score represented as the process P.

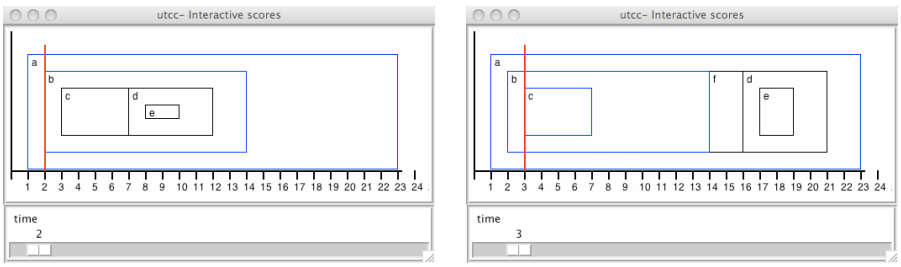
- $[\![P]\!] \models_T \Diamond \exists_{x,d_x,s_x,y,d_y,s_y}(\texttt{box}(x,d_x,s_x) \wedge \texttt{box}(y,d_y,s_y) \wedge \texttt{in}(x,y) \wedge s_x + d_x > s_y + d_y)$: The end time of the box y is less than the end time of the inner box x. I.e., the box y cannot contain x.
- $[\![P]\!] \models_T \Diamond \exists_{d_x,s_x}(\texttt{box}(x,d_x,s_x) \wedge \texttt{play}(x))$: Eventually the structure x is played.

Remark 9.6.1. *For the sake of presentation we only defined here the* `before` *relation. Our model can be straightforwardly extended to support all Allen temporal relations [Allen 1983]. Making use of the* `into` *and* `out` *operations, we can define also the operation* $\texttt{move}(a,b)$ *meaning, move the structure a into the structure b.*

9.7 A Model for Music Improvisation

As described above, in interactive scores the actual musical output may change depending on interactions with a performer, but the framework is not meant for learning from those interactions, nor to change the score (i.e. improvise) accordingly.

Music improvisation provides a complex context of concurrent systems posing great challenges to modeling tools. In music improvisation, partners behave independently but



(a) (b)

Figure 9.6: Example of an Interactive Score Execution

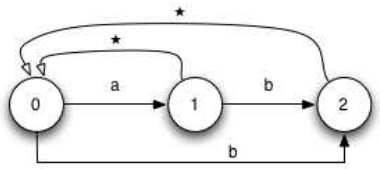


Figure 9.7: A FO automaton for $s = ab$

are constantly interacting with others in controlled ways. The interactions allow building a complex global musical process collaboratively. Interactions become effective when each partner has somehow learned about the possible evolutions of each musical process launched by the others, i.e, their musical *style*. Getting the computer involved in the improvisation process requires learning the musical style of the human interpreter and then playing jointly in the same style. A *style* in this case means some set of meaningful sequences of musical material the interpreter has played. A graph structure called *factor oracle* (FO) is used to efficiently represent this set [Allauzen C. 1999].

A FO is a finite state automaton constructed in an incremental fashion. A sequence of symbols $s = \sigma_1\sigma_2 \dots \sigma_n$ is learned in such an automaton, which states are $0, 1, 2 \dots n$. There is always a transition arrow (called factor link) labeled by the symbol σ_i going from state $i-1$ to state $i, 1 \leq i < n$. Depending on the structure of s, other arrows will be added. Some are directed from a state i to a state j, where $0 \leq i < j \leq n$. These also belong to the set of factor links and are labeled by the symbol σ_j. Some are directed "backwards", going from a state i to a state j, where $0 \leq j < i \leq n$. They are called suffix links, and bear no label (represented as '$\star$' in our processes below). The factor links model a factor automaton, that is every factor p in s corresponds to a unique factor link path labeled by p, starting in 0 and ending in some other state. Suffix links have an important property : a suffix link goes from i to j iff the longest repeated suffix of $s[1..i]$ is recognized in j. Thus suffix links connect repeated patterns of s.

The oracle (see Figure 9.7) is learned on-line. For each new input symbol σ_i, a new state i is added and an arrow from $i-1$ to i is created with label σ_i. Starting from $i-1$, the suffix links are iteratively followed backward, until a state is reached where a factor link with label σ_i originates (going to some state j), or until there is no more suffix links to follow. For each state met during this iteration, a new factor link labeled by σ_i is added from this state to i. Finally, a suffix link is added from i to the state j or to state 0 depending on which condition terminates the iteration. Navigating the oracle in order to generate variants is straightforward : starting in any place, following factor links generates a sequence of labelling symbols that are repetitions of portions of the learned sequence; following one suffix link followed by a factor links creates a recombined pattern sharing a common suffix with an existing pattern in the original sequence. This common suffix is, in effect, the musical context at any given time.

In [Assayag 2006] a `tcc` model of FO is proposed. This model has three drawbacks. Firstly, it (informally) assumes the basic calculus has been extended with general recursion in order to correctly model suffix links traversal. Secondly, it assumes dynamic construction of new variables $\delta_{i\sigma}$ set to the state reached by following a factor link labelled σ from state i. This construction cannot be expressed with the local variable primitive in basic `tcc`. Thirdly, the model assumes a constraint system over both finite domains and finite sets. We use below the expressive power of the abstraction construction in `utcc` to correct all these drawbacks (see Figure 9.8). Furthermore, our model leads to a compact representation of the data structure of the FO based on constraints of the form $\texttt{edge}(x, y, N)$ representing

an arc between node x and y labeled with N.

$$
\begin{array}{lcl}
FO & \stackrel{\text{def}}{=} & Counter \parallel Persistence \\
 & & \parallel !\,(\textbf{abs}\ Note; \texttt{play}(Note))\ \textbf{whenever}\ ready\ \textbf{do}\ Step_1(Note) \\
Counter & \stackrel{\text{def}}{=} & \textbf{tell}(i=1) \parallel !\,(\textbf{abs}\ x; i=x)\,(\textbf{when}\ ready\ \textbf{do}\ \textbf{next}\,\textbf{tell}(i=x+1) \\
 & & \qquad \parallel \textbf{unless}\ ready\ \textbf{next}\,\textbf{tell}(i=x)) \\
Persistence & \stackrel{\text{def}}{=} & !\,(\textbf{abs}\ x,y,z; \texttt{edge}(x,y,z))\ \textbf{next}\,\textbf{tell}(\texttt{edge}(x,y,z)) \\
Step_1(Note) & \stackrel{\text{def}}{=} & \textbf{tell}(\texttt{edge}(i-1,i,Note)) \parallel Step_2(Note, i-1) \\
Step_2(Note,E) & \stackrel{\text{def}}{=} & \textbf{when}\ E=0\ \textbf{do} \\
 & & \quad (\textbf{abs}\ k; \texttt{edge}(E,k,Note))\,(\textbf{tell}(\texttt{edge}(i,k,\star)) \parallel \textbf{next}\,\textbf{tell}(ready)) \\
 & & \quad \parallel \textbf{unless}\ \exists_k\,\texttt{edge}(E,K,Note)\ \textbf{next}\,(\textbf{tell}(ready) \parallel \textbf{tell}(\texttt{edge}(i,0,\star))) \\
 & & \textbf{when}\ E\neq 0\ \textbf{do} \\
 & & \quad (\textbf{abs}\ j; \texttt{edge}(E,j,\star)) \\
 & & \qquad \textbf{when}\ \exists_k\,\texttt{edge}(j,k,Note)\ \textbf{do} \\
 & & \qquad\quad (\textbf{abs}\ k; \texttt{edge}(j,k,Note))\,(\textbf{tell}(\texttt{edge}(i,k,\star)) \parallel \textbf{next}\,\textbf{tell}(ready)) \\
 & & \qquad \parallel \textbf{unless}\ \exists_k\,\texttt{edge}(j,k,Note)\ \textbf{next}\ \textbf{when}\ j\neq 0\ \textbf{do}\ \textbf{tell}(\texttt{edge}(j,i,Note)) \\
 & & \qquad\qquad \parallel Step_2(Note,j)
\end{array}
$$

Figure 9.8: Implementing the FO into `utcc`

Process *Counter* signals when a new played note can be learned. It can be learned when all links for the previous note have already been added to the FO. Process *Persistence* transmits information about already constructed arcs (factor and suffix) to all future time units. Process $Step_1$ adds a factor link from $i-1$ to i labelled with a just played note and launches traversal of suffix links from $i-1$. When state zero is reached by traversing suffix links, process $Step_2$ adds a suffix link from i to a state reached from 0 by a factor link labelled $Note$, if it exists, or from i to state zero, otherwise. For each state k different from zero reached in the suffix links traversal, process $Step_2$ adds factor links labelled $Note$ from k to i.

The inclusion of a new agent in our FO model (e.g. a learner agent for a second performer) entails a new process and new interactions, both with the new process and among the existing ones. In traditional models this usually means major changes in the synchronization scheme, which are difficult to localize and control. In `utcc`, all synchronization is done semantically, through the available information in the store. Each agent would thus have to be incremented with processes testing for the presence of new information (e.g. a factor link with some label in the other agent's FO graph). The new synchronization behavior that this demands is automatically provided by the blocking ask (abstraction) construct.

9.8 Summary and Related Work

In this chapter we showed the application of `utcc` in the modeling and verification of mobile reactive systems in two different emergent areas: Service Oriented Computing and Multimedia Interaction Systems.

The material of this chapter was originally published as [Lopez 2009] and [Olarte 2009b].

Service Oriented Computing. We have argued for `utcc` as a declarative alternative for the analysis of sessions. We presented an encoding of the language for structured communication in [Honda 1998] into `utcc`, as well as an extension of such a language that considers explicitly elements of partial information and session duration. To the best of our knowledge, a unified framework where behavioral and declarative techniques converge has not been proposed before for the analysis of sessions.

Our work has not addressed the typed nature of the HVK language. Roughly speaking, the type discipline in [Honda 1998] ensures a correct "pairing" between complementary components (e.g. session providers and requesters). Our encoding assumes processes to be well-typed with respect to such a discipline. This is because, in our view, declarative techniques should not conflict with operational techniques. Hence, we find it reasonable to assume that a utcc-based analysis of sessions takes places once the type system in [Honda 1998] has ensured a correct pairing.

In this initial effort, we have focused on exploring to what extent temporal logic can provide correctness guarantees at the session level. In a later stage, we expect to undertake a thorough study of the interplay between types, constraints, and temporal formulas in the unified framework CCP provides.

Ongoing work also includes to explore alternative formulations of our encodings. In particular, we would like to determine whether or not they can be expressed in the *monotonic* fragment of utcc, i.e. the fragment without occurrences of **unless** processes. As we have seen, this fragment enjoys more appealing properties. Hence, having encodings of HVK into such a fragment would further support our claims on the convenience of a CCP-based framework for declarative structured communications.

Related Work One approach to combine the declarativeness of constraints and process calculi techniques is represented by a number of works that have extended name-passing calculi with some form of partial information (see, e.g., [Victor 1998, Díaz 1998]). The crucial difference between such a strand of work and CCP-based calculi is that the latter offer a tight correspondence with logic, which greatly broadens the spectrum of reasoning techniques at one's disposal. Recent works similar to ours include cc-pi [Buscemi 2007] and the calculus for structured communication in [Coppo 2008]. Such languages feature elements that resemble much ideas underlying CCP (especially [Buscemi 2007]). The main difference between such works and our approach is that the reasoning techniques they feature are different from logic-based ones. In [Buscemi 2007], a language for Service-Level Agreement (SLA) is proposed, featuring constructs for name-passing, constraint retraction and soft constraints. There, the reasoning techniques are essentially operational. In [Coppo 2008] a language for sessions featuring constraints is proposed. There, the key for analysis is represented by a type system which provides consistency for session execution, much as in the original approach in [Honda 1998].

Multimedia Interaction Systems. We argued for utcc as a declarative framework for modeling and verifying dynamic multimedia interaction systems. We showed that the synchronization mechanism based on entailment of constraints leads to simpler models that scale up when more agents are added. We modeled two non trivial interacting systems. The model proposed for interactive scores in Section 9.6 improved considerably the expressivity of previous models such as [Allombert 2007]. It allows the composer, e.g., to dynamically change the structure of the score according to the information derived from the environment.

It is worth noticing that the variables in utcc are flexible, i.e., they may take different values in each time-unit. In [Manna 1991], it is shown that by universally quantifying on rigid variables and using equality, it is possible to define a counter in FLTL. Assume the following process $P =! (\mathbf{abs}\ u; x = u)\ \mathbf{next\ tell}(x = u + 1)$. If u is a rigid variable, we shall observe that the process P will increase the value of x in each time-unit. Thus, considering explicitly rigid variables in the calculus makes easier, e.g., to define clocks in multimedia interactive systems.

Chapter 10

Abstract Semantics and Static Analysis of `utcc` Programs

In this chapter we propose a semantic framework for the static analysis of `utcc` and `tcc` programs. We consider the denotational semantics for `tcc`, and we extend it to a "collecting" semantics for `utcc`. Relying on this semantics, we formalize a general framework for data flow analyses of `tcc` and `utcc` programs by abstract interpretation techniques [Cousot 1977].

The concrete and abstract semantics we propose are compositional, thus allowing us to reduce the complexity of data flow analyses. Furthermore, the domain of this semantics is simpler with respect to that of the semantics in Chapter 7 . Namely, we shall use sequences of constraints instead of sequences of future-free formulae. This way, we give a precise meaning to `tcc` programs and we effectively approximate the behavior of `utcc` programs.

We show that our method is sound and parametric with respect to the abstract domain. Thus, different analyses can be performed by instantiating the framework. We illustrate for example how it is possible to reuse abstract domains previously defined for logic programming to perform a groundness analysis of a `tcc` program. We show the applicability of this analysis in the context of verification of reactive systems. Furthermore, we make also use of the abstract semantics to automatically exhibit the secrecy flaw in the Needham-Schröder (NS) protocol [Needham 1978] illustrated in Chapter 8.

10.1 Static Analysis and Abstract Interpretation

Static code analysis aims at analyzing properties of a program without actually executing it. The idea is to reason about the semantics of the program which captures the set of all possible outputs it can exhibit when considering an arbitrary input. In the context of `utcc`, recall that the strongest postcondition of a process P, denoted by $sp(P)$, captures the set of sequences that P can output under the influence of an arbitrary environment. Therefore, proving whether P satisfies a given property A, in the presence of any environment, reduces to proving whether $sp(P)$ is a subset of the the set of sequences (outputs) satisfying the property A.

In general, programs properties are undecidable. For example, one may be interested in analyzing when a given program terminates (see e.g., [Giesl 2007, Mesnard 2005]) or determining whether the final value of a variable is in a given interval (see e.g. [Cousot 1977, Bagnara 2007]). In the context of concurrent languages, one may also wonder if there exists a computation leading to a dead lock or if a communication channel is never used (see e.g. [Feret 2005, Garoche 2007, Bodei 1998]).

Abstract interpretation [Cousot 1977, Cousot 1979] is a general theory for approximating the semantics of programs. The idea is to derive a decidable semantics from a concrete one that abstracts away from irrelevant matters. Roughly speaking, in the abstract semantics, concrete properties are replaced by approximated properties modeled by an abstract domain. Because of the approximation, the result is not *complete*, meaning that not all

the properties of the program are discovered. Nevertheless, the result is *sound*, i.e., all the captured properties are satisfied in the concrete semantics.

10.1.1 Static Analysis of Timed CCP Programs

The tcc calculus [Saraswat 1994] was designed for the modeling and verification of reactive systems such as controllers or signal-processing systems. In fact, it has been shown that synchronous data flow languages such as Esterel [Berry 1992] and Lustre [Halbwachs 1991] can be encoded as tcc processes [Saraswat 1994, Tini 1999]. This makes tcc an expressive declarative framework for the modeling and verification of reactive systems, for which it is fundamental to develop tools aiming at helping to develop correct, secure, and efficient programs.

For the analysis of tcc programs we can start building on the frameworks and abstract domains previously defined for Logic Programming, for example in [Cousot 1992, Codish 1999, Armstrong 1998, Comini 2003]. Nevertheless, timed CCP programs pose additional difficulties. Namely, the concurrent, timed nature of the language, and the synchronization mechanism by entailment of constraints (blocking asks). Aiming at statically analyzing utcc as well as tcc programs, we have to consider the additional technical issues due to *mobility*, particularly, the infinite internal computations generated by the **abs** operator in utcc (see Section 3.8).

We shall then proceed as follows. We develop a semantics for tcc and utcc that collects all concrete information required to properly abstract the properties of interest. This semantics is based on closure operators [Scott 1982] over sequences of constraints in the lines of [de Boer 1995b, Saraswat 1994, Nielsen 2002a]. Our semantics is precise for tcc and allows us to effectively approximate the operational semantics of utcc and compositionally describe the behavior of programs. Next, we propose an abstract semantics that approximates the concrete one.

The abstraction we develop proceeds in two-levels. First, we approximate the constraint system leading to an abstract constraint system in the lines of [Falaschi 1997, Zaffanella 1997]. This way, we can capture as "abstract" constraints the properties of interest. Second, as tcc and utcc programs are supposed to run forever, we approximate the output of the program by a finite cut.

The framework we propose is formalized by abstract interpretation techniques and is parametric with respect to the abstract domain. It allows us to exploit also the work done for developing abstract domains for logic programs. Moreover, we can make new analyses for reactive and mobile systems, thus widening the reasoning techniques, available for both, tcc and utcc such as type systems [Hildebrandt 2009], logical characterizations [Mendler 1995, Nielsen 2002a, Olarte 2008c] and semantics [Saraswat 1994, Olarte 2008b, Nielsen 2002a]. Our results then should foster the development of analyzers for different concurrent systems modeled in utcc and its sub-calculi (see [Gupta 1996b, Olarte 2008a] for a survey of applications of CCP-based languages).

Instances of the framework. To show the applicability of our framework, we shall instantiate it in two different scenarios. The first one tailors an abstract domain for groundness and type dependencies analysis in logic programming to perform a groundness analysis of a tcc program. This analysis is proven useful to derive a property of a control system specified in tcc. The second scenario presents an abstraction of the cryptographic constraint system in Definition 8.2.1. We then use the abstract semantics to approximate the behavior of a protocol and exhibit automatically the secrecy flaw illustrated in Section 8.5.

This is done by using a prototypical application of our framework that implements the abstract domain for the verification of secrecy properties.

10.2 Constraint Systems as Information Systems

To develop the abstract interpretation framework for the analysis of `tcc` and `utcc` programs, we need to give a more general notion of constraints than the one we considered in Definition 3.1.1. Namely, up to now we have seen constraints as formulae in first-order logic. It has been useful to formalize the logic characterization of `utcc` processes as formulae in first-order linear-time temporal logic. Nevertheless, for some analysis such as groundness (i.e., determining if a variable is bound to a ground term), we may have as constraints sets of variables. For example, a constraint of the form $c = \{x, y\}$ may represent the information that both x and y are ground variables.

CCP [Saraswat 1991, Saraswat 1993] was originally introduced with a general notion of constraints as information systems [Scott 1982]. Under this definition, a constraint system is a structure $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \texttt{true}, \texttt{false}, \mathit{Var}, \exists, d \rangle$ such that

- $\langle \mathcal{C}, \leq, \sqcup, \texttt{true}, \texttt{false} \rangle$ is a lattice with $\sqcup$ the *lub* operation (representing the logical *and*), and `true`, `false` the least and the greatest elements in $\mathcal{C}$ respectively. Constraints are then the elements in $\mathcal{C}$.
- Var is a denumerable set of variables and for each $x \in \mathit{Var}$ the function $\exists_x : \mathcal{C} \to \mathcal{C}$ is a cylindrification operator satisfying: (1) $\exists_x c \leq c$. (2) If $c \leq d$ then $\exists_x c \leq \exists_x d$. (3) $\exists_x (c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$. (4) $\exists_x \exists_y c = \exists_y \exists_x c$.
- For each $x, y \in \mathit{Var}$, $d_{xy} \in \mathcal{C}$ is a *diagonal element* and it satisfies: (1) $d_{xx} = \texttt{true}$. (2) If z is different from x, y then $d_{xy} = \exists_z (d_{xz} \sqcup d_{zy})$. (3) If x is different from y then $c \leq d_{xy} \sqcup \exists_x (c \sqcup d_{xy})$.

The cylindrification operators model a sort of existential quantification, helpful for defining the **local** operator as we showed in Chapter 3. The diagonal elements are useful to model parameter passing in procedures calls. If $\mathbf{C}$ contains an equality theory, then d_{xy} can be thought as the formulae $x = y$.

Under this definition of constraint system, we say that d *entails* c in $\mathbf{C}$ if and only if $c \leq d$.

The notion of constraint system as first-order formulae in Definition 3.1.1 can be seen as an instance of this more general one. Thus, our results straightforwardly apply when considering the notion of constraint as logic formulae in Chapter 3.

All the notation we have used so far for constraints, terms and substitutions remains the same in this chapter. Nevertheless, to avoid confusion with the abstraction functions usually denoted with α, we shall use s, s' to range over sequences of constraints. This way, we shall write $s \leq s'$ iff $|s| \leq |s'|$ and for all $i \in \{1, \ldots, |s|\}$, $s'(i) \models s(i)$. If $|s| = |s'|$ and for all $i \in \{1, ..., |s|\}$, $s(i) \equiv s'(i)$, we shall write $s \equiv s'$. We shall use $\mathcal{C}^*$ to denote the set of finite sequences of constraints.

10.2.1 Recursion and Parameter Passing

Unlike `utcc`, general recursion cannot be encoded directly in `tcc`. In [Nielsen 2002a], the authors show that only value passing recursion can be defined using the basic constructs. It means, in a call of the form $p(t)$, t is assumed to be a term fixed to a value v, i.e., the current store must entail $t = v$. Furthermore, in a recursive definition of the form

$p(\vec{x}) \stackrel{\texttt{def}}{=} P$, P is restricted to call $p(\cdot)$ at most once and such a call must be within the scope of a **next** operator. The reason for such a restriction is to avoid infinitely or unboundedly many recursive calls of $p(\vec{x})$ within the same time interval.

For this reason, to broaden the applicability of our framework, we shall consider tcc programs with recursive definitions.

Definition 10.2.1 (Timed CCP programs)**.** *Given a set of procedure declarations $\mathcal{D}$, a (u)tcc program takes the form $\mathcal{D}.P$ where P is a (u)tcc process. For every procedure name, we assume that there exists one and only one corresponding declaration in $\mathcal{D}$.*

We remind the reader that tcc is a subcalculus of utcc and that recursive definition does not add any expressive power to utcc as we explained in Section 3.3.1.

Operational Semantics. We slightly modify the rules of the operational semantics in Table 3.1 to be consistent with the definition of constraint system in this chapter. We also add the rule $\mathrm{R}_{\mathrm{CALL}}$ to deal with recursive definitions in tcc (see Table 10.1). The structural congruence relation "$\equiv$" is the same as in Definition 3.4.1 and the Future function F for the rule $\mathrm{R}_{\mathrm{OBS}}$ is the same as in Definition 3.4.2.

In the rule $\mathrm{R}_{\mathrm{CALL}}$ we make use of the diagonal elements to model parameter passing as standardly done in CCP [Saraswat 1991]. In this rule,

$$\Delta^{\vec{x}}_{\vec{y}} P = (\mathbf{local}\,\vec{a})\,(!\,\mathbf{tell}(d_{\vec{x}\vec{a}}) \parallel (\mathbf{local}\,\vec{y})\,(!\,\mathbf{tell}(d_{\vec{a}\vec{y}}) \parallel P))$$

where the variables in $\vec{a}$ are assumed to occur neither in the declaration nor in the process P, and $d_{\vec{x}\vec{y}}$ denotes the constraint $\bigsqcup_{1\leq i\leq|\vec{x}|} d_{x_i y_i}$. Roughly speaking, $\Delta^{\vec{x}}_{\vec{y}}$ equates the formal parameters $\vec{x}$ and the actual parameters $\vec{y}$ (see [Saraswat 1993]).

The notions of Observables and input-output behavior are the same as in Section 3.7 considering the new definition of the internal reduction relation in Table 10.1.

Strongest Postcondition. Since we are considering here the operational semantics which only outputs basic constraints, we do not require the notion of fixed formulae to define the strongest postcondition of a process as in Chapter 4. We then define the strongest postcondition for the operational semantics as standardly done in tcc and CCP [Saraswat 1991, de Boer 1995b, Nielsen 2002a].

Definition 10.2.2 (Strongest Postcondition)**.** *Let $io(\cdot)$ be the input-output relation in Definition 3.7.1. Given a utcc process P, the strongest postcondition of P, denoted by $sp(P)$, is defined as the set $\{s \mid (s,s) \in io(P)\}$.*

We can think of $sp(P)$ as the set of sequences that P can output under the influence of an arbitrary environment. Therefore, proving whether P satisfies a given property A, in the presence of any environment, reduces to proving whether $sp(P)$ is a subset of the set of sequences (outputs) satisfying the property A.

10.3 A Denotational model for tcc and utcc

As we explained before, the strongest postcondition relation fully captures the behavior of a process considering any possible output under an arbitrary environment. In this section we develop a denotational model for the strongest postcondition. The semantics is the basis for the abstract interpretation framework we shall develop in the next section.

$$\mathrm{R_{TELL}}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \sqcup c\rangle}$$

$$\mathrm{R_{PAR}}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle}$$

$$\mathrm{R_{LOC}}\ \frac{\langle P, c \sqcup (\exists \vec{x} d)\rangle \longrightarrow \langle P', c' \sqcup (\exists \vec{x} d)\rangle}{\langle (\mathbf{local}\, \vec{x}; c)\, P, d\rangle \longrightarrow \langle (\mathbf{local}\, \vec{x}; c')\, P', d \sqcup \exists \vec{x} c'\rangle}$$

$$\mathrm{R_{UNL}}\ \frac{d \models c}{\langle \mathbf{unless}\ c\ \mathbf{next}\ P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle}$$

$$\mathrm{R_{REP}}\ \frac{}{\langle !\, P, d\rangle \longrightarrow \langle P || \mathbf{next}\, !\, P, d\rangle}$$

$$\mathrm{R_{ABS}}\ \frac{d \models c[\vec{t}/\vec{x}] \quad |\vec{t}| = |\vec{x}| \quad [\vec{t}/\vec{x}] \text{ is admissible.}}{\langle (\mathbf{abs}\ \vec{x}; c)\, P, d\rangle \longrightarrow \left\langle P[\vec{t}/\vec{x}] || (\mathbf{abs}\ \vec{x}; c \sqcup \vec{x} \neq \vec{t}\,)\, P, d\right\rangle}$$

$$\mathrm{R_{STR}}\ \frac{\gamma_1 \longrightarrow \gamma_2}{\gamma_1' \longrightarrow \gamma_2'} \quad \text{if } \gamma_1 \equiv \gamma_1' \text{ and } \gamma_2 \equiv \gamma_2'$$

$$\mathrm{R_{CALL}}\ \frac{p(\vec{y}) \stackrel{\mathtt{def}}{=} P \in \mathcal{D}}{\langle p(\vec{x}), d\rangle \longrightarrow \left\langle \Delta^{\vec{x}}_{\vec{y}} P, d\right\rangle}$$

$$\mathrm{R_{OBS}}\ \frac{\langle P, c\rangle \longrightarrow^{*} \langle Q, d\rangle \not\longrightarrow}{P \stackrel{(c,d)}{\Longrightarrow} F(Q)}$$

Table 10.1: Operational Semantics for tcc and utcc considering constraint systems as partial information systems. $\not\models$ and admissibility of $[\vec{t}/\vec{x}]$ are defined in Convention 3.1.1.

Our semantics is built on the closure operator semantics for tcc in [Saraswat 1994, Nielsen 2002a] and specifies compositionally the strongest postcondition relation in Definition 10.2.2. Unlike the semantics in Chapter 7, the semantics we present here is more appropriate for the data-flow analysis due to its simpler domain based on sequences of constraints instead of sequences of temporal formulae. In Section 10.6 we elaborate more on the differences between both semantics.

Roughly speaking, the semantics is based on a (continuous) immediate consequence operator $T_{\mathcal{D}}$, which computes in a bottom-up fashion the *interpretation* of each procedure definition $p(\vec{x}) \stackrel{\mathtt{def}}{=} P$ in $\mathcal{D}$. Such an interpretation is given in terms of the set of the quiescent sequences for $p(\vec{x})$.

Compositional Semantics. Let *ProcHeads* denote the set of process names with their formal parameters. Recall that $\mathcal{C}^{\omega}$ stands for the set of infinite sequences of constraints. We shall call *Interpretations* the set of functions in the domain $\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^{\omega})$. The semantics is defined as a function $[\![\cdot]\!] : (\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^{\omega})) \to (\mathit{Proc} \to \mathcal{P}(\mathcal{C}^{\omega}))$ which given an interpretation I, associates to each process a set of sequences of constraints.

$\mathrm{D}_{\mathrm{SKIP}}$	$[\![\mathbf{skip}]\!]_I$	$=$	$\mathcal{C}^\omega$
$\mathrm{D}_{\mathrm{TELL}}$	$[\![\mathbf{tell}(c)]\!]_I$	$=$	$\{d.s \in \mathcal{C}^\omega \mid d \models c\}$
$\mathrm{D}_{\mathrm{PAR}}$	$[\![P \parallel Q]\!]_I$	$=$	$[\![P]\!]_I \cap [\![Q]\!]_I$
$\mathrm{D}_{\mathrm{NEXT}}$	$[\![\mathbf{next}\, P]\!]_I$	$=$	$\{d.s \in \mathcal{C}^\omega \mid s \in [\![P]\!]_I\}$
$\mathrm{D}_{\mathrm{UNL}}$	$[\![\mathbf{unless}\ c\ \mathbf{next}\ P]\!]_I$	$=$	$\{d.s \in \mathcal{C}^\omega \mid d \not\models c \text{ and } s \in [\![P]\!]_I\} \cup \{d.s \in \mathcal{C}^\omega \mid d \models c\}$
$\mathrm{D}_{\mathrm{REP}}$	$[\![!\, P]\!]_I$	$=$	$\{s \in \mathcal{C}^\omega \mid \text{ for all } s'', s' \text{ s.t. } s = s''.s',\ s' \in [\![P]\!]_I\}$
$\mathrm{D}_{\mathrm{LOC}}$	$[\![(\mathbf{local}\, \vec{x}; c)\, P]\!]_I$	$=$	$\{s \in \mathcal{C}^\omega \mid \text{ there exists an } \vec{x}\text{-variant } s' \text{ of } s \text{ s.t. } s'(1) \models c \text{ and } s' \in [\![P]\!]_I\}$
$\mathrm{D}_{\mathrm{ASK}}$	$[\![\mathbf{when}\ c\ \mathbf{do}\ P]\!]_I$	$=$	$\{d.s \in \mathcal{C}^\omega \mid d \models c \text{ and } d.s \in [\![P]\!]_I\} \cup \{d.s \in \mathcal{C}^\omega \mid d \not\models c\}$
$\mathrm{D}_{\mathrm{ABS}}$	$[\![(\mathbf{abs}\ \vec{x}; c)\, P]\!]_I$	$=$	$\bigcap_{\vec{t} \in \mathcal{T}^{\lvert\vec{x}\rvert}} [\![(\mathbf{when}\ c\ \mathbf{do}\ P)[\vec{t}/\vec{x}]]\!]_I$
$\mathrm{D}_{\mathrm{CALL}}$	$[\![p(\vec{x})]\!]_I$	$=$	$I(p(\vec{x}))$

Table 10.2: Semantic Equations for `tcc` and `utcc` constructs. In $\mathrm{D}_{\mathrm{ABS}}$, if $|\vec{x}| = 0$ then $\mathcal{T}^{|\vec{x}|}$ is defined as $\{\varepsilon\}$

The semantic equations are given in Table 10.2. They are similar to those in Figure 7.1. The main difference is that each equation is parametric on an interpretation I. This interpretation is used to give meaning to the calls of procedures (Rule $\mathrm{D}_{\mathrm{CALL}}$).

Notice that here we follow the semantic equation for the abstraction operator $(\mathbf{abs}\ x; c)\, P$ based on the representation of this operator as a parallel composition $\prod_{\vec{t} \in \mathcal{T}^{|\vec{x}|}} (\mathbf{when}\, c\, \mathbf{do}\, P)[\vec{t}/\vec{x}]$ where $\mathcal{T}$ denotes the set of terms in the underlying constraint system (see Section 7.2.1).

Concrete Domain. The domain of the denotation is $\mathbb{E} = (E, \subseteq^c)$ where $E = \mathcal{P}(\mathcal{C}^\omega)$ and $\subseteq^c$ is a Smyth-like ordering defined as follows: Let $X, Y \in E$ and $\lesssim$ be the preorder s.t $X \lesssim Y$ iff for all $y \in Y$, there exists $x \in X$ s.t. $x \leq y$. $X \subseteq^c Y$ iff $X \lesssim Y$ and $(Y \lesssim X$ implies $Y \subseteq X)$. The bottom of $\mathbb{E}$ is then $\mathcal{C}^\omega$ (the set of all the sequences). We do not consider the empty set to be part of the domain. Then, the top element is the singleton $\{\mathtt{false}^\omega\}$ (since `false` is the greatest element in $(\mathcal{C}, \leq)$).

We note that the Hoare power domain is not suitable for our construction since the bottom element would be the empty set. Then, all intersection (due to a parallel composition) will collapse.

Formally, the semantics is defined as follows:

Definition 10.3.1 (Concrete Semantics)**.** *Let $[\![\cdot]\!]_I$ be defined as in Table 10.2. The seman-*

tics of a program $\mathcal{D}.P$ is defined as the least fixed point of the continuous operator:

$$T_{\mathcal{D}}(I)(p(\vec{y})) = [\![\Delta^{\vec{x}}_{\vec{y}} P']\!]_I \text{ if } p(\vec{x}) \stackrel{\text{def}}{=} P' \in \mathcal{D}$$

We shall use $[\![P]\!]$ to represent $[\![P]\!]_{lfp(T_{\mathcal{D}})}$

Let us exemplify the least fixed point construction above.

Example 10.3.1. *Assume two constraints $\mathtt{out}_a(\cdot)$ and $\mathtt{out}_b(\cdot)$,intuitively representing outputs of names on two different channels a and b. Let $\mathcal{D}$ be the following procedure definitions*

$$\begin{array}{rcll} \mathcal{D} & = & p() \stackrel{\text{def}}{=} & \mathbf{tell}(\mathtt{out}_a(x)) \parallel \mathbf{next}\,\mathbf{tell}(\mathtt{out}_a(y)) \\ & & q() \stackrel{\text{def}}{=} & (\mathbf{abs}\ z; \mathtt{out}_a(z))\,\mathbf{tell}(\mathtt{out}_b(z)) \parallel \mathbf{next}\, q() \\ & & r() \stackrel{\text{def}}{=} & p() \parallel q() \end{array}$$

The procedure $p()$ outputs on channel a the variables x and y in the first and second time units respectively. The procedure $q()$ resends on channel b every message received on channel a. Starting from the bottom interpretation $I_\perp$ (assigning $\mathcal{C}^\omega$ to each name procedure), the semantics of $r()$ is obtained as follows

$$\begin{array}{lll} I_1 : & p \to \{c.c'.s \mid c \models \mathtt{out}_a(x) \text{ and } c' \models \mathtt{out}_a(y)\} \\ & q \to \{c_1.s \mid c_1 \models \mathtt{out}_a(t) \text{ implies } c_1 \models \mathtt{out}_b(t)\} \\ & r \to \mathcal{C}^\omega \cap \mathcal{C}^\omega = \mathcal{C}^\omega \\ I_2 : & p \to I_1(p) \\ & q \to \{c_1.c_2.s \mid c_i \models \mathtt{out}_a(t) \text{ implies } c_i \models \mathtt{out}_b(t) \ i{=}1{,}2\} \\ & r \to I_1(p) \cap I_1(q) \\ \ldots & \\ I_\omega : & p \to I_1(p) \\ & q \to \{s \mid (s(i) \models \mathtt{out}_a(t) \text{ imp. } s(i) \models \mathtt{out}_b(t) \text{ for } i > 0\} \\ & r \to I_\omega(p) \cap I_\omega(q) \end{array}$$

where t denotes any term. In words, if $s \in [\![r()]\!]$ then $s(1) \models \mathtt{out}_a(x)$, $s(2) \models \mathtt{out}_a(y)$ and for $i \geq 1$, if $s(i) \models \mathtt{out}_a(t)$ then $s(i) \models \mathtt{out}_b(t)$

10.3.1 Semantic Correspondence

In this section we prove the semantic correspondence between the operational and the semantics in Definition 10.3.1. Before that, recall that unlike tcc, some utcc processes may exhibit infinite behavior during a time unit due to the abstraction operator (see Section 3.8). Considering this fact, it may be the case that sequences in the input-output behavior (and then in the strongest postcondition in Definition 10.2.2) are *finite* or even the empty sequence ε. Therefore, unlike the results in Chapter 7 relating the symbolic strongest postcondition and the denotational semantics, here we relate the outputs of a process and the subsequences of its denotation.

Before stating the soundness theorem, we require a similar result to that in Proposition 7.3.1 but using the notion of Strongest Postcondition in Definition 10.2.2.

Proposition 10.3.1. *Let $P = (\mathbf{abs}\ \vec{x}; c)\, Q$ and s be a sequence of constraints. The following statements are equivalent.*

- $s \in sp(P)$.
- $\mathcal{T}' = \{\vec{t_i} \mid s(1) \models c[\vec{t_i}/\vec{x}]\} \subseteq_{fin} \mathcal{T}^{|\vec{x}|}$ *and for all $\vec{t} \in \mathcal{T}'$ admissible for $\vec{x}$, $s \in sp(Q[\vec{t}/\vec{x}])$.*

Proof. Let $P = (\mathbf{abs}\ \vec{x}; c)\, Q$ and $s = c_1.c_2.c_3....$ By alpha conversion we assume that $\vec{x} \notin fv(s)$.

($\Rightarrow$) Assume that $s \in sp_s(P)$. Then, there exists $P_1 = P'_1, P'_2...., P'_i$ such that

$$P = P_1 \xrightarrow{(c_1,c_1)} P_2 \xrightarrow{(c_2,c_2)} ....P_i \xrightarrow{(c_i,c_i)}$$

Let $\vec{t} \in \mathcal{T}^{|\vec{x}|}$ be an arbitrary term such that $s(1) \models c[\vec{t}/\vec{x}]$. Let $Q_1 = Q^1_1 = Q[\vec{t}/\vec{x}]$ and $P_1 = P^1_1$. Since $c_1 \models c[\vec{t}/\vec{x}]$, by rule $\mathrm{R_{ABS}}$ we must have a derivation

$$\langle P^1_1, c_1\rangle \longrightarrow^* \langle P^i_1 \parallel Q^1_1, c_1\rangle \longrightarrow^* \langle P^m_1 \parallel Q^n_1, c_1\rangle \not\longrightarrow$$

for some $P^2_1, .., P^m_1$ and $Q^2_1, .., Q^n_1$. Since $P^1_1 = P_1$ and $P_1 \xrightarrow{(c_1,c_1)} P_2$, we have $P_2 \equiv F(P^m_1 \parallel Q^n_1)$ where F is the future function in Definition 3.4.2. By the semantics of the parallel composition we can verify that

$$Q_1 \xrightarrow{(c_1,c_1)} Q_2 \xrightarrow{(c_2,c_2)} Q_3 \xrightarrow{(c_3,c_3)} \dots$$

Since $Q_1 = Q[\vec{t}/\vec{x}]$ we conclude $s \in sp(Q[\vec{t}/\vec{x}])$.

($\Leftarrow$) Let $\mathcal{T}' = \{\vec{t_i} \mid s(1) \models c[\vec{t_i}/\vec{x}]\} \subseteq_{fin} \mathcal{T}^{|\vec{x}|}$ and assume that for any $\vec{t} \in \mathcal{T}'$ we have $s \in sp(Q[\vec{t}/\vec{x}])$, i.e., we have a derivation of the form

$$Q[\vec{t}/\vec{x}] = Q_1 \xrightarrow{(c_1,c_1)} Q_2 \xrightarrow{(c_2,c_2)} Q_3 \xrightarrow{(c_3,c_3)} ...$$

By the rule $\mathrm{R_{ABS}}$ we know that

$$\langle P, c_1\rangle \longrightarrow^* \langle P' \parallel \prod_{\vec{t_i} \in \mathcal{T}'} Q[\vec{t_i}/\vec{x}], c_1\rangle \longrightarrow^* \langle P' \parallel \prod_{\vec{t_i} \in \mathcal{T}'} Q'_i[\vec{t_i}/\vec{x}], c_1\rangle \not\longrightarrow$$

We conclude by noticing that if none of the $Q[\vec{t_i}/\vec{x}]$ above can add new information to s, it must be the case that $P \xrightarrow{(s,s)}$ and then $s \in sp(P)$.

□

The following theorem shows that if a (finite) sequence s is in the strongest postcondition, then there exists a infinite sequence s' in the denotation such that s is a prefix of s'.

Theorem 10.3.1 (Soundness)**.** *Let $[\![\cdot]\!]$ be as in Definition 10.3.1. Given a program $\mathcal{D}.P$, if $s \in sp(P)$ then there exists s' s.t. $s.s' \in [\![P]\!]$.*

Proof. The proof proceeds by induction on the structure of the process P. All the cases but $P = (\mathbf{abs}\ \vec{x}; c)\, Q$ are the same as in `tcc` and proven in [de Boer 1995b, Saraswat 1994, Nielsen 2002a]. We then only prove the case for the abstraction operator.

Assume that $P = (\mathbf{abs}\ \vec{x}; c)\, Q$ and $s \in sp(P)$. Recall that $\mathtt{false}^\omega$ is quiescent for any process. As a mean of contradiction assume that $s' = s.\mathtt{false}^\omega \notin [\![P]\!]$. Then, there exists $\vec{t}$ s.t. $s' \notin [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]$. Then, it must be the case that $s'(1) \models c[\vec{t}/\vec{x}]$ and $s' \notin [\![Q[\vec{t}/\vec{x}]]\!]$. Since $s \in sp(P)$ and $s(1) \models c[\vec{t}/\vec{x}]$, by Proposition 10.3.1, $s \in sp(Q[\vec{t}/\vec{x}])$. By inductive hypothesis $s \in [\![Q[\vec{t}/\vec{x}]]\!]$ then a contradiction. □

Similar to the Theorem 7.3.2, the completeness theorem holds only for the local independent and abstracted-unless free fragment of `utcc`.

Theorem 10.3.2 (Completeness)**.** *Let $\mathcal{D}.P$ be a locally independent and abstracted-unless free program s.t. $s \in [\![P]\!]$. For all prefixes s' of s, if there exists s'' s.t. $(s', s'') \in io(P)$ then $s' \equiv s''$, i.e., $s' \in sp(P)$.*

Proof. The proof proceeds by induction on the structure of the process P. All the cases but $P = (\mathbf{abs}\ \vec{x}; c)\, Q$ are the same as in `tcc` and proven in [de Boer 1995b, Saraswat 1994, Nielsen 2002a]. We then only prove the case for the abstraction operator.

Let $P = (\mathbf{abs}\ \vec{x}; c)\, Q$. By extensiveness we know that if $(s', s'') \in io(P)$ then $s' \leq s''$. As a mean of contradiction assume that $s \in [\![P]\!]$ and there exists a prefix s' of s s.t. $(s', s'') \in io(P)$ and $s' < s''$ (i.e., $s' \notin sp(P)$). Then, by Proposition 10.3.1, there exists $\vec{t}$ s.t. $s'(1) \models c[\vec{t}/\vec{x}]$ and $s' \notin sp(Q[\vec{t}/\vec{x}])$. By inductive hypothesis, there is no a sequence s'' s.t. $s = s'.s'' \in [\![Q[\vec{t}/\vec{x}]]\!]$. Given that s' is a prefix of s, $s(1) \models c[\vec{t}/\vec{x}]$. Since $s \in [\![P]\!]$ and $s(1) \models c[\vec{t}/\vec{x}]$, by Equation $\mathrm{D}_{\mathrm{ABS}}$ we have $s \in [\![Q[\vec{t}/\vec{x}]]\!]$. Thus a contradiction. □

10.4 Abstract Interpretation Framework

In this section we develop the abstract interpretation framework [Cousot 1992] for the analysis of `utcc` programs. The framework is based on the above denotational semantics, thus allowing for a compositional analysis of `utcc` (and then `tcc`) programs. The abstraction proceeds in two-levels: (1) we abstract the constraint system and then (2) we abstract the infinite sequences of *abstract* constraints by a finite cut. The abstraction in (1) allows us to reuse the most popular abstract domains previously defined for logic programming. Adapting those domains, it is possible to perform, e.g., groundness, freeness, type and suspension analyses of `tcc` and `utcc` programs. Furthermore, it allows us to restrict the set of terms to be considered in the Equation $\mathrm{D}_{\mathrm{ABS}}$. Thus, we can even approximate the output of a non-well terminated process as we show in Section 10.5.3. On the other hand, the abstraction in (2) along with (1) allow for computing the approximated output of the program in a finite number of steps.

10.4.1 Abstract Constraint Systems

Let us recall some notions from [Falaschi 1997] and [Zaffanella 1997].

Definition 10.4.1 (Abstract C.S. and Descriptions)**.** *Given two constraint systems*

$$\begin{array}{lcl} \mathbf{C} & = & \langle \mathcal{C}, \leq \sqcup, \mathtt{true}, \mathtt{false}, \mathit{Var}, \exists, d \rangle \\ \mathbf{A} & = & \langle \mathcal{A}, \leq^{\alpha} \sqcup^{\alpha}, \mathtt{true}^{\alpha}, \mathtt{false}^{\alpha}, \mathit{Var}, \exists^{\alpha}, d^{\alpha} \rangle \end{array}$$

a description $(\mathcal{C}, \alpha, \mathcal{A})$ consists of an abstract domain $(\mathcal{A}, \leq^{\alpha})$ and a monotonic abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$. We lift α to sequences of constraints in the obvious way.

We shall use c_κ, d_κ to range over constraints in $\mathbf{A}$ and s_κ, s'_κ to range over sequences in $\mathcal{A}^\omega$ and $\mathcal{A}^*$. Let $\models^{\alpha}$ be defined as in the concrete counterpart, i.e. $c_\kappa \leq^{\alpha} d_\kappa$ iff $d_\kappa \models^{\alpha} c_\kappa$. The set of abstract terms is denoted by $\mathcal{T}_\kappa$ and ranged by $t_\kappa, t'_\kappa ...$

Following standard lines in [Falaschi 1997, Zaffanella 1997] we impose the following restrictions over α:

Definition 10.4.2 (Correctness)**.** *Let $\alpha : \mathcal{C} \to \mathcal{A}$ be monotonic. We say that $\mathbf{A}$ is* upper correct *w.r.t the constraint system $\mathbf{C}$ if for all $c \in \mathcal{C}$ and $x, y \in \mathcal{V}$: (1) $\alpha(\exists_x c) = \exists^{\alpha}_x \alpha(c)$. (2) $\alpha(d_{xy}) = d^{\alpha}_{xy}$. (3) $\alpha(c \sqcup d) \models^{\alpha} \alpha(c) \sqcup^{\alpha} \alpha(d)$. Let $\alpha_t : \mathcal{T} \to \mathcal{T}_\kappa$ be the term-abstraction structurally based on α. Given the sequence of variables $\vec{x}$ and $\vec{t}, \vec{t'} \in \mathcal{T}^{|\vec{x}|}$, (4) $\alpha(c[\vec{t}/\vec{x}]) = \alpha(c[\vec{t'}/\vec{x}])$ whenever $\alpha_t(\vec{t}) = \alpha_t(\vec{t'})$.*

Conditions (1), (2) and (3) relate the cylindrification, diagonal and *lub* operators of both constraints systems. Condition (4) is only necessary to have a safe approximation of the **abs** operator in `utcc`, but it is not required when analyzing `tcc` programs. It informally says that substituting for terms mapped to the same abstract term, must lead to the same abstract constraint.

In the example below we illustrate an abstract domain for the groundness analysis of `tcc` programs. Here we give just an intuitive description of it. We shall elaborate more on this domain and its applications in Section 10.5.1.

Example 10.4.1. *Let the Herbrand constraint system (Hcs) [Saraswat 1991] be the concrete domain. In Hcs, a first-order language $\mathcal{L}$ with equality is assumed. The entailment relation is that one expects from equality, e.g., $[x|y] = [a|z]$ must entail $x = a$ and $y = z$. Terms, as usual, are variables, constants and functions applied on terms. As abstract constraint system, let constraints be predicates of the form iff$(x, [])$ meaning that x is a ground variable. Abstract terms are variables and the special term g meaning "ground". In this setting, $\alpha(x = [a]) =$ iff $(x, [])$ (i.e., x is a ground variable). Furthermore $\alpha_t(a) = \alpha_t(b) = g$. Therefore, by Condition (4) in Definition 10.4.2, $\alpha((x = [y])[a/y]) = \alpha((x = [y])[b/y]) =$ iff$(x, [])$.*

We conclude this section by defining when an "abstract" constraint approximates a concrete one.

Definition 10.4.3 (Approximations)**.** *Let* **A** *be* upper correct *w.r.t* **C** *and $(\mathcal{C}, \alpha, \mathcal{A})$ be a description. Given $d_\kappa = \alpha(d)$, we say that d_κ is the best approximation of d. Furthermore, for all $c_\kappa \leq^\alpha d_\kappa$ we say that c_κ approximates d and we write $c_\kappa \propto d$. This definition is extended to sequences of constraints in the obvious way.*

10.4.2 Abstract Semantics

Starting from the semantics in Section 10.3, we develop here an abstract semantics which approximates the observable behavior of a program and is adequate for modular data-flow analysis. We focus our attention on a special class of abstract interpretations obtained from what we call a *sequence abstraction* mapping possibly infinite sequences of (abstract) constraints into finite ones.

Definition 10.4.4 (Sequence Abstraction)**.** *A sequence abstraction $\tau : \mathcal{A}^\omega \cup \mathcal{A}^* \to \mathcal{A}^*$ is an anti-extensive ($\tau(s_\kappa) \leq^\alpha s_\kappa$) and monotonic operator. We lift τ to sets of sequences in the obvious way: $\tau(S_\kappa) = \{s_\kappa \mid s_\kappa = \tau(s'_\kappa)$ and $s'_\kappa \in S\}$.*

A simple albeit useful instance of the abstraction τ is the *sequence*(k) cut. This abstraction approximates a sequence by projecting it to its first k elements, e.g., *sequence*$(2)(s_\kappa) = s_\kappa(1).s_\kappa(2)$.

Abstract Domain. Given a description $(\mathcal{C}, \alpha, \mathcal{A})$, we choose as concrete domain $\mathbb{E} = (E, \subseteq^c)$ as defined in Section 10.3. The abstract domain is $\mathbb{A} = (A, \subseteq^\alpha)$ where $A = \mathcal{P}(\mathcal{A}^*)$ and $\subseteq^\alpha$ is defined similarly to $\subseteq^c$: Let $X, Y \in A$ and $\lesssim^\alpha$ be the preorder s.t. $X \lesssim^\alpha Y$ iff for all $y \in Y$, there exists $x \in X$ s.t. $x \leq^\alpha y$. $X \subseteq^\alpha Y$ iff $X \lesssim^\alpha Y$ and ($Y \lesssim^\alpha X$ implies $Y \subseteq X$). The bottom and top of this domain are, similar to the concrete domain, $\mathcal{A}^*$ and $\{\texttt{false}^\alpha . \texttt{false}^\alpha ...\}$ respectively.

We require $\mathbb{A}$ to be noetherian (i.e., there are no infinite ascending chains). This guarantees that the fixed point of the abstract semantics can be reached in a finite number of iterations.

The semantic equations are given in Table 10.3. We shall dwell a little upon the description of the rules A_{ASK}, A_{ABS} and A_{UNL}. The other cases are easy.

For the case of A_{ASK}, we follow [Zaffanella 1997, Falaschi 1993, Falaschi 1997] for the abstract semantics of the synchronization (ask) operator in CCP. Intuitively, the Equation A_{ASK} says that if the abstract computation proceeds, then every concrete computation it approximates proceeds too. This is formalized by the relation $d_\kappa \models\!\!\!\models_{\mathcal{A}} c$, meaning that the abstract constraint d_κ entails c if all concrete constraint approximated by d_κ entails c.

Definition 10.4.5. *Given $d_\kappa \in \mathcal{A}$ and $c \in \mathcal{C}$, $d_\kappa \models\!\!\!\models_{\mathcal{A}} c$ iff for all $c' \in \mathcal{C}$ s.t. $d_\kappa \propto c'$, $c' \models c$.*

In Equation A_{ABS}, we compute the intersection over the abstract terms ($\mathcal{T}_\kappa$) and we replace $\vec{x}$ with a concrete term $\vec{t'}$ s.t. $\alpha_t(\vec{t'}) = \vec{t_\kappa}$. Notice that it may be the case that there exists $\vec{t_1}$, $\vec{t_2}$ s.t. $\alpha_t(\vec{t_1}) = \alpha_t(\vec{t_2}) = \vec{t_\kappa}$. Using property (4) in Definition 10.4.2, we can show that the choice of the concrete term is irrelevant.

Proposition 10.4.1. *Let $[\![\cdot]\!]^\tau_X$ be as in Table 10.3 and $\vec{t_1}$, $\vec{t_2}$ be concrete terms different from $\vec{x}$ s.t. $|\vec{x}| = |\vec{t_1}| = |\vec{t_2}|$ and $\alpha_t(\vec{t_1}) = \alpha_t(\vec{t_2})$. For every sequence s_κ s.t. $\vec{x} \notin fv(s_\kappa)$, $s_\kappa \in [\![P[\vec{t_1}/\vec{x}]]\!]^\tau_X$ iff $s_\kappa \in [\![P[\vec{t_2}/\vec{x}]]\!]^\tau_X$.*

Proof. Let $\vec{t_1}$, $\vec{t_2} \in \mathcal{T}^{|\vec{x}|}$ and assume that $\vec{t_1} \neq \vec{t_2}$. Assume also that $s_\kappa \in [\![P[\vec{t_1}/\vec{x}]]\!]^\tau_X$. One can show that $s_\kappa \in [\![(\mathbf{local}\,\vec{x})\,(P \parallel !\,\mathbf{tell}(\vec{x} = \vec{t_1}))]\!]^\tau_X$. By Equations D_{LOC} and D_{PAR}, there exists s'_κ $\vec{x}$-variant of s_κ s.t. $s'_\kappa \in [\![P]\!]^\tau_X$ and $s'_\kappa \in [\![!\,\mathbf{tell}(\vec{x} = \vec{t_1})]\!]^\tau_X$. By Property (4) in Definition 10.4.2, $s'_\kappa \in [\![!\,\mathbf{tell}(\vec{x} = \vec{t_2})]\!]^\tau_X$. We conclude $s_\kappa \in [\![P[\vec{t_2}/\vec{x}]]\!]^\tau_X$. □

Abstract Semantics for the unless operator. One could think of defining the abstract semantics of the **unless** operator similarly to that of the **when** operator as follows:

$$[\![\mathbf{unless}\ c\ \mathbf{next}\ P]\!]^\tau_X = \tau(\{d_\kappa.s_\kappa \mid d_\kappa \not\models\!\!\!\models_{\mathcal{A}} c \text{ and } s_\kappa \in [\![P]\!]^\tau_X) \cup \tau(\{d_\kappa.s_\kappa \mid d_\kappa \models\!\!\!\models_{\mathcal{A}} c)$$

Nevertheless, this equation leads to a non safe approximation of the concrete semantics. This is because from $d_\kappa \not\models\!\!\!\models_{\mathcal{A}} c$ we cannot conclude that $d \not\models c$ where $\alpha(d) = d_\kappa$. To see this, take $Q = \mathbf{unless}\ c\ \mathbf{next}\ P$ and d such that $d \models c$. Then $d.\,\mathtt{true}^\omega \in [\![Q]\!]$. Take c' such that $c' \not\models c$ and $c'_\kappa = \alpha(c') \leq^\alpha \alpha(d) = d_\kappa$. Then, $d_\kappa \propto c'$ and $d_\kappa \not\models\!\!\!\models_{\mathcal{A}} c$. If P is not the process **skip**, we have $d_\kappa.\,\mathtt{true}^* \notin [\![Q]\!]^\tau$.

Defining $d_\kappa \not\models\!\!\!\models_{\mathcal{A}} c$ as true iff $c' \not\models c$ for all c' approximated by d_κ does not solve the problem. This is because under this definition, $d_\kappa \not\models\!\!\!\models_{\mathcal{A}} c$ would not hold for any d_κ and c: `false` entails all the concrete constraints and it is approximated for every abstract constraint.

Therefore, we cannot give a better (safe) approximation of the semantics of $Q = \mathbf{unless}\ c\ \mathbf{next}\ P$ than $\tau(\mathcal{A}^\omega)$, i.e. $[\![Q]\!]^\tau_X = [\![\mathbf{skip}]\!]^\tau_X$ (Rule A_{UNL}).

Abstract Semantics. We define formally the abstract semantics as follows:

Definition 10.4.6. *Let $[\![\cdot]\!]^\tau_X$ be as in Table 10.3. The abstract semantics of a program $\mathcal{D}.P$ is defined as the least fixed point of the following continuous semantic operator:*

$$T^\alpha_{\mathcal{D}}(X)(p(\vec{x})) = [\![(\Delta^{\vec{y}}_{\vec{x}} P')]\!]^\tau_X \text{ if } p(\vec{y}) \stackrel{\mathtt{def}}{=} P' \in \mathcal{D}$$

We shall use $[\![P]\!]^\tau$ to denote $[\![P]\!]^\tau_{lfp(T^\alpha_{\mathcal{D}})}$

$\mathrm{A}_{\mathrm{SKIP}}$	$[\![\mathbf{skip}]\!]^\tau_X$	$=$	$\tau(\mathcal{A}^\omega)$
$\mathrm{A}_{\mathrm{TELL}}$	$[\![\mathbf{tell}(c)]\!]^\tau_X$	$=$	$\tau(\{d_\kappa.s_\kappa \in \mathcal{A}^\omega \mid d_\kappa \models^\alpha \alpha(c)\})$
$\mathrm{A}_{\mathrm{PAR}}$	$[\![P \parallel Q]\!]^\tau_X$	$=$	$[\![P]\!]^\tau_X \cap [\![Q]\!]^\tau_X$
$\mathrm{A}_{\mathrm{NEXT}}$	$[\![\mathbf{next}\, P]\!]^\tau_X$	$=$	$\tau(\{d_\kappa.s_\kappa \in \mathcal{A}^* \mid s_\kappa \in [\![P]\!]^\tau_X\})$
$\mathrm{A}_{\mathrm{UNL}}$	$[\![\mathbf{unless}\ c\ \mathbf{next}\ P]\!]^\tau_X$	$=$	$\tau(\mathcal{A}^\omega)$
$\mathrm{A}_{\mathrm{REP}}$	$[\![!\, P]\!]^\tau_X$	$=$	$\tau(\{s_\kappa \in \mathcal{A}^* \mid$ for all s'_κ, w_κ s.t. $s_\kappa = w_\kappa.s'_\kappa$, $s'_\kappa \in [\![P]\!]^\tau_X\})$
$\mathrm{A}_{\mathrm{LOC}}$	$[\![(\mathbf{local}\, \vec{x}; c)\, P]\!]^\tau_X$	$=$	$\tau(\{s_\kappa \in \mathcal{A}^* \mid$ there exists a $\vec{x}$-variant s'_κ of s_κ s.t. $s'_\kappa(1) \models^\alpha \alpha(c)$ and $s'_\kappa \in [\![P]\!]^\tau_X\})$
$\mathrm{A}_{\mathrm{ASK}}$	$[\![\mathbf{when}\ c\ \mathbf{do}\ P]\!]^\tau_X$	$=$	$\tau(\{d_\kappa.s_\kappa \in \mathcal{A}^* \mid d_\kappa \not\models_{\mathcal{A}} c\}) \cup$ $\tau(\{d_\kappa.s_\kappa \in \mathcal{A}^* \mid d_\kappa \models_{\mathcal{A}} c$ and $d_\kappa.s_\kappa \in [\![P]\!]^\tau_X\})$
$\mathrm{A}_{\mathrm{ABS}}$	$[\![(\mathbf{abs}\ \vec{x}; c)\, P]\!]^\tau_X$	$=$	$\bigcap_{\vec{t_\kappa} \in \mathcal{T}_\kappa^{\lvert\vec{x}\rvert}} [\![(\mathbf{when}\ c\ \mathbf{do}\ P)[\vec{t'}/\vec{x}]]\!]^\tau_X$ where $\alpha_t(\vec{t'}) = \vec{t_\kappa}$
$\mathrm{A}_{\mathrm{CALL}}$	$[\![p(\vec{x})]\!]^\tau_X$	$=$	$X(p(\vec{x}))$

Table 10.3: Abstract denotational semantics for utcc. $\models_{\mathcal{A}}$ in Definition 10.4.5

10.4.3 Soundness of the Approximation

This section proves the correctness of the abstract semantics in Definition 10.4.6. We first establish a Galois insertion between the concrete and the abstract domains. From [Zaffanella 1997, Proposition 3], we deduce the following:

$$\begin{aligned} \underline{\alpha}(E) &:= \tau(\{\alpha(s) \mid s \in E\}) \\ \gamma(A) &:= \{s \mid \tau(\alpha(s)) \in A\} \end{aligned}$$

We have used $\underline{\alpha}$ to avoid confusion with α in $(\mathcal{C}, \alpha, \mathcal{A})$. We can lift in the standard way to abstract interpretations [Cousot 1992] the approximation induced by the above abstraction. Let $I : \mathit{ProcHeads} \to E$, $X : \mathit{ProcHeads} \to A$ and p a procedure name. Then

$$\begin{aligned} \underline{\alpha}(I)(p) &:= \tau(\{\alpha(s) \mid s \in I(p)\}) \\ \gamma(X)(p) &:= \{s \mid \tau(\alpha(s)) \in X(p)\} \end{aligned}$$

The next theorem proves that the concrete computations are safely approximated by the abstract semantics.

Theorem 10.4.1 (Soundness of the approximation)**.** *Let* **A** *be upper correct with respect to* **C**, $(\mathcal{C}, \alpha, \mathcal{A})$ *be a description and* τ *be a sequence abstraction. Let* $[\![\cdot]\!]$ *and* $[\![\cdot]\!]^\tau$ *be respectively as in Definitions 10.3.1 and 10.4.6. Given a* utcc *program* $\mathcal{D}.P$*, if* $s \in [\![P]\!]$ *then* $\tau(\alpha(s)) \in [\![P]\!]^\tau$.

Proof. The proof proceeds by induction on the structure of P. Assume that $s \in [\![P]\!]$ and that $s_\kappa = \tau(\alpha(s))$. In each case we shall prove that $s_\kappa \in [\![P]\!]^\tau$.

- $P =$ **skip**. This case is trivial since $\tau(\mathcal{A}^\omega)$ approximates every possible concrete computation.
- $P =$ **tell**(c). If $s \in [\![P]\!]$ then $s(1) \models c$. Let $s_\kappa = \tau(\alpha(s))$. Then, it must be the case that $s_\kappa(1) \models^\alpha \alpha(c)$ and then $\tau(\alpha(s)) \in [\![P]\!]^\tau$.
- $P = Q \parallel R$. We must have that $s \in [\![Q]\!]$ and $s \in [\![R]\!]$. By inductive hypothesis we know that $s_\kappa \in [\![Q]\!]^\tau$ and $s_\kappa \in [\![R]\!]^\tau$ and then, $s_\kappa \in [\![Q \parallel R]\!]^\tau$.
- $P =$ **next** Q. Let $s' = s(2).s(3)....$, $s'_\kappa = s_\kappa(2).s_\kappa(3)....$, We know that $s' \in [\![Q]\!]$ and by inductive hypothesis $s'_\kappa \in [\![Q]\!]^\tau$. We then conclude $s_\kappa \in [\![P]\!]^\tau$.
- $P =$ **unless** c **next** Q. This case is trivial since $\tau(\mathcal{A}^\omega)$ approximates every possible concrete computation.
- $P =! Q$. We now that every suffix of s' of s is in $[\![Q]\!]$. By induction the corresponding suffix of s'_κ of s_κ is in $[\![Q]\!]^\tau$ and we conclude $s_\kappa \in [\![P]\!]^\tau$.
- $P =$ **when** c **do** Q. Assume that $s(1) \models c$ and then $s \in [\![Q]\!]$. We can have either $s_\kappa \models_{\mathcal{A}} c$ or $s_\kappa \not\models_{\mathcal{A}} c$. In the first case, since $s \in [\![Q]\!]$ by induction $s_\kappa \in [\![Q]\!]^\tau$ and then $s_\kappa \in [\![P]\!]^\tau$. If $s_\kappa \not\models_{\mathcal{A}} c$, then trivially $s_\kappa \in [\![P]\!]^\tau$.

  Assume now that $s(1) \not\models c$. Then, it must be the case that $s_\kappa(1) \not\models_{\mathcal{A}} c$ and then trivially $s_\kappa \in [\![P]\!]^\tau$.
- $P =$ (**abs** $\vec{x}; c) Q$. If $s(1) \models c[\vec{t}/\vec{x}]$ for some $\vec{t} \in \mathcal{T}$ then $s \in [\![$**when** c **do** $Q[\vec{t}/\vec{x}]]\!]$. By an analysis similar to the case of $P =$ **when** c **do** Q we can show that $s_\kappa \in [\![$**when** c **do** $Q[\vec{t}/\vec{x}]]\!]^\tau$. We conclude by noticing that if there exist $\vec{t}$ and $\vec{t'}$ s.t. $\alpha_t(\vec{t_\kappa}) = \alpha'_t(\vec{t_\kappa})$, by Proposition 10.4.1 it must be the case that $s_\kappa \in [\![$**when** c **do** $Q[\vec{t}/\vec{x}]]\!]^\tau$ iff $s_\kappa \in [\![$**when** c **do** $Q[\vec{t'}/\vec{x}]]\!]^\tau$.

□

10.5 Applications

This section describes two specific abstract domains as instances of our framework. Firstly, we tailor two abstract domains from logic programming to perform a groundness and a type analysis of a `tcc` program. We then apply this analysis in the verification of a reactive system in `tcc`. Secondly, we abstract the cryptographic constraint system in Definition 8.2.1. We then use the abstract semantics to approximate the behavior of a protocol and exhibit automatically the secrecy flaw illustrated in Section 8.5.

10.5.1 Groundness Analysis

In logic programming, one useful analysis is groundness. It aims at determining if a variable will always be bound to a ground term. This information can be used, e.g., for optimization in the compiler (to remove code for suspension checks at runtime) or as base for other data flow analyses such as independence analysis, suspension analysis, etc. Here we illustrate a groundness analysis for a `tcc` program. To this end, we shall use as concrete domain the Herbrand constraint system (Hcs) [Saraswat 1991] (see Example 10.4.1).

Assume the following procedure definitions:

$$
\begin{array}{lcl}
gen_a(x) & \stackrel{\text{def}}{=} & (\textbf{local}\, x')\,(!\,\textbf{tell}(x = [a|x']) \parallel \\
& & \quad \textbf{when}\ go_a\ \textbf{do next}\ gen_a(x') \parallel \\
& & \quad \textbf{when}\ stop_a\ \textbf{do}\ !\,\textbf{tell}(x' = [\,]))
\end{array}
$$

$$
\begin{array}{lcl}
gen_b(x) & \stackrel{\text{def}}{=} & (\textbf{local}\, x')\,(!\,\textbf{tell}(x = [b|x']) \parallel \\
& & \quad \textbf{when}\ go_b\ \textbf{do next}\ gen_b(x') \parallel \\
& & \quad \textbf{when}\ stop_b\ \textbf{do}\ !\,\textbf{tell}(x' = [\,]))
\end{array}
$$

$$
\begin{array}{lcl}
append(x, y, z) & \stackrel{\text{def}}{=} & \textbf{when}\ x = [\,]\ \textbf{do}\ !\,\textbf{tell}(y = z) \parallel \\
& & \textbf{when}\ \exists_{x', x''}(x = [x'\ |x'']\,)\ \textbf{do} \\
& & \quad (\textbf{local}\, x', x'', z')\,(!\,\textbf{tell}(x = [x' \mid x'']) \parallel \\
& & \qquad\qquad !\,\textbf{tell}(z = [x' \mid z']) \parallel \\
& & \qquad\qquad \textbf{next}\ append(x'', y, z'))
\end{array}
$$

The process $gen_a(x)$ adds to the stream x an "a" when the environment provides go_a as input. Under input $stop_a$, it terminates the stream binding its tail to the empty list. Let x_go_a and x_stop_a be two distinct variables different from x and x', and go_a and $stop_a$ be the constraints $x\_go_a = []$ and $x\_stop_a = []$. The process gen_b can be explained similarly. The process $append(x, y, z)$ binds z to the concatenation of x and y.

The Abstract Domain. We shall use *Pos* [Armstrong 1998] as abstract domain for the groundness analysis. In *Pos*, positive propositional formulae represent groundness dependencies among variables. Elements in the domain are ordered by logical implication. Let α_g be defined over equations in normal form as:

$$\alpha_g(x = t) = \text{iff}(x, fv(t))$$

For instance, $\alpha_g(x = [y|z]) = \text{iff}(x, \{y, z\})$ representing the propositional formula $x \Leftrightarrow (y \wedge z)$ meaning, x is ground if and only if y and z are ground.

Notice that *Pos* does not distinguish between the empty list and a list of ground terms, i.e., $d_\kappa = \alpha_g(x = []) = \alpha_g(x = [a]) = \text{iff}(x, \{\})$. Therefore, we have $d_\kappa \not\models_{\mathcal{A}} x = []$ (see Definition 10.4.5).

Type Dependency Analysis. We can improve the accuracy of our analysis by using the abstract domain in [Codish 1994] to derive information about type dependencies on terms. The abstraction is defined as follows:

$$\alpha_T(x = t) = \begin{cases} list(x, x_s) & \text{if} \quad t = [y \mid x_s] \text{ for some } y \\ nil(x) & \text{if} \quad t = [] \end{cases}$$

Informally, $list(x, x_s)$ means x is a list iff x_s is a list and $nil(x)$ means x is the empty list. If x is a list we write $list(x)$. Elements in the domain are ordered by logical implication.

Let $\mathbf{A_g} = \langle \mathcal{A}, \leq^\alpha \ \sqcup^\alpha, \texttt{true}^\alpha, \texttt{false}^\alpha, Var, \exists^\alpha, d^\alpha \rangle$ be the abstract constraint system obtained from the reduced product ([Cousot 1992]) of the previous abstract domains. Elements $g, g' ... \in \mathcal{A}$ are tuples $\langle c_\kappa, d_\kappa \rangle$ where c_κ corresponds to groundness information and d_κ to type dependency information. The abstraction function is defined as expected, i.e., $\alpha(c) = g = \langle \alpha_g(c), \alpha_T(c) \rangle$. The operations $\sqcup^\alpha$, $\exists^\alpha$ correspond to logical conjunction and existential quantification over the components of the tuple. The diagonal element d_{xy} corresponds to $\langle x = y, x = y \rangle$. Finally, $\langle c_\kappa, d_\kappa \rangle \leq^\alpha \langle c'_\kappa, d'_\kappa \rangle$ if $c'_\kappa \Rightarrow c_\kappa$ and $d'_\kappa \Rightarrow d_\kappa$.

$$
\begin{array}{l}
micCtrl(Error, Button) \stackrel{\texttt{def}}{=} \\
\quad (\textbf{local}\, E', B', e, b) \\
\qquad !\,\textbf{tell}(Error = [e \mid E'] \sqcup Button = [b|B']) \\
\qquad \parallel \textbf{when}\ \texttt{on} \sqcup \texttt{open}\ \textbf{do} \\
\qquad\quad !\,\textbf{tell}(e = yes \sqcup E' = [] \sqcup b = stop) \\
\qquad \parallel \textbf{when}\ \texttt{off}\ \textbf{do tell}(e = no) \parallel \textbf{next}\, micCtrl(E', B') \\
\qquad \parallel \textbf{when}\ \texttt{closed}\ \textbf{do tell}(e = no) \parallel \textbf{next}\, micCtrl(E', B')
\end{array}
$$

Figure 10.1: `tcc` model for a microwave controller.

Let $\tau = sequence(\kappa)$ and $g_1.g_2....g_\kappa \in [\![Gen_a(x)]\!]^\tau$. By a derivation similar to that of Example 10.3.1, if there exists $i \in \{1, .., \kappa\}$ such that $g_i \models_{\mathcal{A}} stop_a$, one can show that there exists $\vec{x'} = x'_0, x'_1, ..., x'_i$ such that

$$
g_i \models^{\alpha} \exists_{\vec{x'}} \left\langle \begin{array}{l} \mathrm{iff}(x, x'_0) \sqcup \bigsqcup_{0<j<i} \mathrm{iff}(x'_j, \{x'_{j+1}\}) \sqcup \mathrm{iff}(x'_i, \{\}) \\ list(x, x'_0) \sqcup \bigsqcup_{0<j<i} list(x'_j, x'_{j+1}) \sqcup nil(x'_i) \end{array} \right\rangle ,
$$

Thus, if $g_i \models_{\mathcal{A}} stop_a$ we can deduce that x is a list and x is a ground variable.

Let $s_k = [\![Gen_a(x) \parallel Gen_b(y) \parallel append(x, y, z)]\!]^\tau$. If there exist $i, j \leq \kappa$ s.t. $s_\kappa(i) \models_{\mathcal{A}} stop_a$ and $s_\kappa(j) \models_{\mathcal{A}} stop_b$, we can show that for $l \geq max(i, j)$, the variables x, y and z are list of ground elements. More precisely,

$$
s_\kappa(l) \models^{\alpha} \langle \mathrm{iff}(x, []) \sqcup \mathrm{iff}(y, []) \sqcup \mathrm{iff}(z, []), list(x) \sqcup list(y) \sqcup list(z) \rangle
$$

10.5.2 Analysis of Reactive Systems

The works in [Saraswat 1994, Tini 1999] show that synchronous data flow languages such as Esterel [Berry 1992] and Lustre [Halbwachs 1991] can be encoded as `tcc` processes. This makes `tcc` an expressive declarative framework for the modeling and verification of reactive systems. Here we show how our framework can provide additional reasoning techniques in `tcc` for the verification of such systems: We shall use the groundness analysis developed in Section 10.5.1 to verify if a simplified version of a control system for a microwave complies with its intended behavior. Namely, the door must be closed when it is turned on. Let us introduce first the model of such a system.

Example 10.5.1 (Control System)**.** *Assume a simple control system for a microwave checking that the door must be closed when it is turned on. Otherwise, it must emit an error signal. The specification in* `tcc` *of this system is depicted in Figure 10.1.*

In this `tcc` *program, constraints of the form $X = [e|X']$ asserts that X is a list with head e and tail X'. This way, the process micCtrl binds Error to a list ended by "yes" when the microwave was turned on and the door was open at the same interval of time. Furthermore, the constant stop is added into the list Button signaling the environment that the microwave must be powered off.*

Similarly to the example in Section 10.5.1, we assume `on`, `off`, `closed` and `open` be respectively the constraints $on = [], off = [], close = []$ and $open = []$ for variables on, off, $close$ and $open$ different from E and E'. The symbols yes and $stop$ denote constant symbols.

Our analysis consists in determining when the variable *Error* is bound to a ground term. If the system is correct, it must happen when the the door is open and the microwave is turned on.

Let $\tau = sequence(\kappa)$ for a given κ. We can verify that if $s_\kappa \in [\![micCtrl(Error, Button)]\!]^\tau$ and $s_\kappa(i) \models_{\mathcal{A}} (\texttt{open} \sqcup \texttt{on})$ then $s_\kappa(i) \models^\alpha \langle iff(Error, []), list(Error)\rangle$, i.e., *Error* is a ground variable.

We then conclude that the system effectively binds the list Error to a ground term whenever the system reaches an inconsistent state.

10.5.3 Analyzing Secrecy Properties

In this section we show how the abstract semantics allows us to approximate the behavior of a security protocol modeled in SCCP (see Chapter 8). We shall propose an abstraction of the Security Constraint System in Definition 8.2.1 and then, with the help of a prototypical tool, we exhibit automatically the secrecy flaw in the Needham-Schröder (NS) protocol [Needham 1978] illustrated in Section 8.5.

Recall that the rules PAIR and ENC in the secure constraint system (Definition 8.2.1) generates infinite behavior due to the infinitely many messages (constraints) they can entail. For example, let $P = (\textbf{abs}\ x; \texttt{out}(x))\, Q$. If the current store is $\texttt{out}(m)$, by the rule PAIR, P must execute $Q[\{m, m\}/x]$, $Q[\{m, \{m, m\}\}/x]$ and so on.

To deal with this state explosion problem, the number of messages to be considered can be bound (see e.g. [Song 2001]). We formalize this with the following abstraction.

Definition 10.5.1 (Abstract secure cons. system)**.** *Let $\mathcal{M}$ be the set of (terms) messages in the constraint system in Definition 8.2.1 and $lg : \mathcal{M} \to \mathbb{N}$ be defined as:*

$$lg(m) = \begin{cases} 0 & \text{if} \quad m \in \mathcal{P} \cup \mathcal{K} \cup Var \\ 1 + lg(m_1) + lg(k) & \text{if} \quad m = \{m_1\}_k \\ 1 + lg(m_1) + lg(m_2) & \text{if} \quad m = (m_1, m_2) \end{cases}$$

Let cut_κ be the following term abstraction

$$cut_\kappa(m) = \begin{cases} m & \text{if} \ \ lg(m) \leq \kappa \\ m_\top & \text{otherwise} \end{cases}$$

with $m_\top \notin \mathcal{M}$ (representing all the messages with length greater than κ) . Let $\mathcal{C}$ be as in Definition 8.2.1 and $(\mathcal{C}, \alpha, \mathcal{A})$ be a description where $\alpha(\texttt{out}(m)) = \texttt{out}(cut_\kappa(m))$.

10.5.3.1 Secrecy Analysis

To illustrate our approach, we shall use the NS protocol that we modeled in *SCCP* in Section 8.5.

Assume the configuration involving the principals $\mathcal{P} = \{A, B, C\}$ and where the private key of C has been leaked. Recall that this configuration can be modeled as follows:

$$\mathsf{NS} = \mathsf{Init}(A, C) \parallel_{X \in \mathcal{P}} \mathsf{Resp}'(X) \parallel \mathsf{Spy}$$

where

$$\begin{array}{lcl}
\mathsf{Init}(A,B) & = & !\,\mathbf{new}(m) \\
 & & \quad \mathbf{out}(\{(m,A)\}_{pub(B)}). \\
 & & \quad\quad \mathbf{in}\ (x)[\{(m,x,B)\}_{pub(A)}].\mathbf{out}(\{x\}_{pub(B)}).\mathbf{nil} \\
 & & \\
\mathsf{Resp}'(B) & = & \mathbf{in}\ (x,u)[\{(x,u)\}_{pub(B)}]. \\
 & & \quad \mathbf{new}(n)(\mathbf{out}(\{m,n,B\}_{pub\,(u)}).\mathbf{nil}) \parallel \\
 & & \quad\quad\quad !\,\mathbf{in}\ [n].\mathbf{out}(attack).\mathbf{nil}) \\
\mathsf{Spy} & = & \|_{A\in\mathcal{P}}\ !\,\mathbf{out}(A).\mathbf{nil} \\
 & & \|_{A\in\mathcal{P}}\ !\,\mathbf{out}(pub(A)).\mathbf{nil} \\
 & & \|_{A\in Bad}\ !\,\mathbf{out}(priv(A)).\mathbf{nil}
\end{array}$$

The abstraction cut_3 and $\tau = sequence(2)$ allows us to verify the following:

$$\text{if } s_\kappa \in [\![NS]\!]^\tau \text{ then } s_\kappa(2) \models^\alpha \mathtt{out}(attack)$$

meaning that *NS* leads to a secrecy attack. Notice the importance of having here a finite cut of the messages (terms) generated by the constraint system to compute $[\![NS]\!]^\tau$. This allows us to restrict the set of terms considered by the **abs** operator and over-approximate its behavior.

10.5.4 A prototypical implementation

We have implemented our framework and the abstract domain for secrecy analysis in a prototype developed in Oz (`http://www.mozart-oz.org/`). This tool is described at

`http://www.lix.polytechnique.fr/~colarte/prototype/`

and allows the user to compute the least element of the abstract semantics of a process P. The current implementation supports constraints as those used in the cryptographic constraint system (e.g., predicates of the form $\mathtt{out}(enc(x, pub(y)))$). It implements the $sequence(\kappa)$ and $cut_{\kappa'}$ abstractions where κ and κ' are parameters specified by the user. We started by implementing the secrecy analysis since one of the most appealing application of the `utcc` calculus is the modeling and verification of security protocols. Our goal is to develop (or adapt from existing implementation) previously defined domains for logic programs such as those used in Section 10.5.1. This then will provide a valuable tool for the analysis of `tcc` and `utcc` programs.

The reader may find in the URL above a deeper description of the tool and some examples. Furthermore, we provide the program excerpts to compute the output of the secrecy analysis for the Needham-Schroeder Protocol [Lowe 1996]. We also illustrate a similar analysis for the Denning-Sacco key distribution protocol [Denning 1981]

10.6 Summary and Related Work

In this chapter we presented a general abstract interpretation framework for the static analysis of `utcc` and `tcc` programs. We built on a compositional semantics based on closure operators on sequences of constraints. We first approximated the constraint system and then we computed a finite cut on the infinite sequences of constraints produced by the concrete semantics. As an application of our framework, we showed how to adapt domains from logic programming to perform a groundness analysis on `tcc` programs. We then applied this analysis to verify a property of a control system modeled in `tcc`. Finally, we showed

that the abstract semantics allows us to automatically exhibit the secrecy flaw illustrated in Section 8.5.

The material of this chapter was originally published as [Falaschi 2009, Falaschi 2007].

Related Work Several frameworks and abstract domains for the analysis of logic programs have been defined (see e.g. [Cousot 1992, Codish 1999, Armstrong 1998]). Those works differ from ours since they do not deal with the temporal behavior and synchronization mechanisms present in tcc-based languages. On the contrary, since our framework is parametric with respect to the abstract domain, it can benefit from those works.

Unlike the semantics based on sequences of future-free formulae in Chapter 7, the semantics here presented turned out to be more appropriate to develop our abstract interpretation framework. Firstly, the inclusion relation between the strongest postcondition and the semantics is verified for the whole language (Theorem 10.3.1) – in the semantics of Chapter 7 this inclusion is verified only for the locally-independent and abstracted-unless free fragment–. Secondly, this semantics makes use of the entailment relation over constraints rather than the more involved entailment over first-order linear-time temporal formulae. This shall ease the implementation of tools based on the framework. Finally, our semantics allows us to capture the behavior of tcc programs with recursion. This is not possible with the semantics in Chapter 7 which was built only for utcc programs where recursion can be encoded (see Section 3.3.2).

The framework we propose here provides the theoretical basis for building tools for the data-flow analyses of utcc and tcc programs whose verification and debugging are not trivial due to their concurrent nature and synchronization mechanisms. We have shown for example, how to analyze groundness and how to detect mistakes in safety critical applications, such as control systems and embedded systems.

Our results should foster the development of analyzers for different concurrent systems modeled in utcc and its sub-calculi. We plan to perform freeness, suspension, type and independence analyses among others. It is well known that this kind of analyses have many applications, e.g. for code optimization in compilers, for improving run-time execution, and for approximated verification.

We believe that the framework proposed here can also help to develop new analyses for other languages for reactive systems (e.g. Esterel [Berry 1992]), which can be translated into tcc [Tini 1999, Saraswat 1994] and for languages featuring mobile behavior as the π-calculus [Milner 1992b, Sangiorgi 2001]. For the latter, many analyses have been already defined, see e.g. [Feret 2005, Garoche 2007]. As future work, it would be interesting to see if it is possible to carry out similar analyses in our framework for suitable fragments of π that can be encoded into utcc (see e.g., Chapter 9 where we encode a π-based language for structured communication into utcc).

Chapter 11

Concluding Remarks

We conclude this dissertation by summarizing its contributions and describing possible directions for future research. The reader can find a more detailed summary, related and future work in the final section of each chapter.

11.1 Overview

In this dissertation we studied a declarative model for the specification of *mobile reactive* systems based on the Saraswat's Concurrent Constraint Programming model [Saraswat 1993]. To do this, we introduced Universal Timed CCP (`utcc`), an extension of `tcc` [Saraswat 1994] with the ability to express mobile behavior.

We added to `tcc` an *abstraction* operator of the form $(\mathbf{abs}\ \vec{x}; c)\, P$ that represents a *temporary parametric ask* process. We illustrated how the interplay of this operator and the local operator $(\mathbf{local}\, \vec{x}; c)\, Q$ allows for the communication of local names, i.e., mobility.

Since abstractions in `utcc` may generate infinitely many internal reductions in the operational semantics (Chapter 3), we endowed the language with a symbolic semantics (Chapter 4) that uses temporal constraints to represent finitely a possible infinite number of substitutions. We proved that for all processes this semantics produces a finite number of internal reductions during a time-unit.

The relevance of the model we proposed in this dissertation is that it complies with two criteria that distinguish CCP from other formalisms for concurrency. Namely, (1) a declarative view of processes as logic formulae and (2) determinism. For (1), we proved in Chapter 5 a strong correspondence of `utcc` processes with formulae in Pnueli's First-Order Linear-Time Temporal Logic (FLTL) [Manna 1991]. This then allowed us to perform reachability analysis of systems modeled in `utcc`. As for (2), we proved the outputs of a process to be equivalent when considering the same input.

The criteria (1) and (2) above allowed us to develop a rich theory for `utcc` with applications in different fields of Computer Science. For instance, it allowed us to exhibit a secrecy flaw of a security protocol (Chapter 8), to give a declarative characterization of sessions (Chapter 9) and to verify minimal conditions to be satisfied for a multimedia interaction system to avoid raise conditions in execution time (Chapter 9).

The deterministic nature of the `utcc` calculus allowed us to give a simple and elegant characterizations of `utcc` processes as closure operators (Chapter 7). We use this semantic characterization in Chapter 8 to give a closure operator semantics to a language for security. This way, we brought new semantic insights into the verification of security protocols.

We also studied the expressiveness of the `utcc` calculus with respect to its predecessor `tcc`. We showed that, unlike `tcc` where processes can be represented as *finite-state* Büchi automata, a very simple constraint system is enough to encode Minsky machines into `utcc` (Chapter 6). We also showed that `utcc` can compositionally encode the call-by-name lambda calculus.

As a compelling application of the underlying theory in `utcc`, we showed in Chapter 6 that the monadic fragment of FLTL without equality nor function symbols is strongly

incomplete, and then, undecidable its validity problem. Our decidability result is insightful since it fills a gap on the decidability study of monadic FLTL: This result refutes a decidability conjecture for FLTL in [Valencia 2005]. It also justifies the restriction imposed in previous decidability results on the quantification of flexible-variables [Merz 1992].

Finally, we provided an abstract interpretation framework as basis for the static analysis of `utcc` processes. We showed that the abstract semantics allows us to approximate the behavior of non well-terminated processes as those arising in the verification of security protocols. Since the framework we proposed is parametric on the abstract domain, we showed that it is possible to reuse abstract domains previously defined in logic programming to analyze `utcc` programs.

11.2 Future Directions

The following are, in the author's opinion, some interesting directions for future work.

Non-determinism. In several application domains it is convenient to be able to specify non-deterministic behavior. Take for example the language for structured communication studied in Section 9.1. One may be interested in specifying competing services where a client can choose among several servers that offer the same service. As it was pointed out in Observation 9.3.1, this behavior cannot be modeled in `utcc` due to its deterministic nature.

In process calculi the non-determinism can arise from an explicit *choice* construct as in CCS [Hoare 1985], the π-calculus [Milner 1992b, Sangiorgi 2001] and the `ntcc` calculus [Nielsen 2002a]. It is also possible that the parallel operator introduces non-determinism (linearity) as in the case of the π-calculus and Linear CCP [Fages 2001, Saraswat 1992].

It would be interesting to study how restricted forms of non-determinism and linearity can be introduced in `utcc` to broaden its applicability. The works in [Nielsen 2002a, Fages 2001, Saraswat 1992, de Boer 1995a, Falaschi 1997, de Boer 1997] may bring some ideas on how to deal with the semantics and the logic correspondence issues arisen when non-determinism is added to CCP-based languages.

FLTL Correspondence and Decidability of FLTL. We plan to extend the proof system in [Nielsen 2002a] to consider the `utcc` abstraction operator and then, to cope with judgements of the form $P \vdash_T A$ where A is a past-free formula. The meaning of this judgment is that every possible output of P is a model for the formula A. Notice that in [Nielsen 2002a] the underlying logic is CLTL (a temporal logic where formulae are interpreted on sequences of constraints). The semantics of the underlying logic in `utcc` is given in terms of sequences of states as described in Section 2.4. Both semantics are related as it was shown in [Valencia 2005, Lemma 5.4].

The formula obtained from the encoding of Minsky machines in Chapter 6 is clearly not monodic (i.e., a temporal subformula has more than one free variable –[Degtyarev 2002]) . It would be interesting to find the minimum number of distinct free variables that can occur in a FLTL formula (or in a TLV-like logic) to obtain undecidability as in [Degtyarev 2002] for the case of TLP (see related work in Section 6.6).

Type Systems. In [Hildebrandt 2009] a type system for `utcc` was proposed to avoid processes of the form $(\mathbf{abs}\ x, y; \mathtt{out}(x, y))\, P$ where both, the channel name (x) and the message sent (y) are quantified (see Sections 3.9 and 8.7). It would be interesting to continue the study of type disciplines for CCP-based languages. For example, one could

define a type system in the lines of [Honda 1998] for `HVK-T` (see Section 9.1) better suited to the nature of `utcc` processes. Studying new type disciplines and their relation with reasoning techniques based on temporal logic seems also to be an interesting research field.

Abstract Interpretation Framework. The framework proposed in Chapter 10 is intended to be the basis for developing static analyzers for CCP programs. Here much work remains to be done at the implementation level. From the theoretical point of view, it seems to be a challenge to develop abstract domains for CCP that cope with the loss of information due to the synchronization mechanism based on entailment of constraints. It would be also interesting to explore abstract debugging techniques (see e.g., [Comini 1999, Falaschi 2007]) for `utcc` programs using the semantics of Chapter 10.

Bibliography

[Abadi 1990] Martín Abadi. *Corrigendum: The Power of Temporal Proofs.* Theor. Comput. Sci., vol. 70, no. 2, page 275, 1990. 67

[Abadi 1997] Martín Abadi and Andrew D. Gordon. *A Calculus for Cryptographic Protocols: The Spi Calculus.* In Fourth ACM Conference on Computer and Communications Security. ACM Press, 1997. 33, 105, 111, 112

[Abadi 2005] Martín Abadi and Bruno Blanchet. *Analyzing Security Protocols with Secrecy Types and Logic Programs.* Journal of the ACM, vol. 52, no. 1, 2005. 111

[Abramsky 1993] Samson Abramsky and C.-H. Luke Ong. *Full Abstraction in the Lazy Lambda Calculus.* Inf. Comput., vol. 105, no. 2, pages 159–267, 1993. 80

[Allauzen C. 1999] Raffinot M. Allauzen C. Crochemore M. *Factor oracle: a new structure for pattern matching'.* In Proc. of SOFSEM'99. LNCS, 1999. 124, 128

[Allen 1983] James F. Allen. *Maintaining knowledge about temporal intervals.* Commun. ACM, vol. 26, no. 11, 1983. 125, 127

[Allombert 2006] A. Allombert, G. Assayag, M. Desainte-Catherine and C. Rueda. *Concurrent constraints models for interactive scores.* In proceedings of SMC ' 06, 2006. 124

[Allombert 2007] A. Allombert, G. Assayag and M. Desainte-Catherine. *A system of interactive scores based on Petri nets.* In proceedings of SMC ' 07, 2007. 123, 124, 125, 130

[Amadio 2003] Roberto M. Amadio, Denis Lugiez and Vincent Vanackère. *On the symbolic reduction of processes with cryptographic functions.* Theor. Comput. Sci., vol. 290, no. 1, 2003. 105, 111

[Aranda 2007] J. Aranda, Gerard Assayag, Carlos Olarte, Jorge A. Pérez, Camilo Rueda, Mauricio Toro and Frank D. Valencia. *An Overview of FORCES: An INRIA Project on Declarative Formalisms for Emergent Systems.* In ICLP'09. Springer, 2007. 17

[Aranda 2009] Jesús Aranda, Frank D. Valencia and Cristian Versari. *On the Expressive Power of Restriction and Priorities in CCS with Replication.* In FOSSACS, volume 5504 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2009. 86

[Armstrong 1998] T. Armstrong, K. Marriott, P. Schachte and H. Søndergaard. *Two classes of Boolean functions for dependency analysis.* Science of Computer Programming, vol. 31, no. 1, 1998. 132, 144, 148

[Assayag 2006] G. Assayag, S. Dubnov and C. Rueda. *A Concurrent Constraints Factor Oracle Model for Music Improvisation.* In CLEI 06, 2006. 124, 128

[Baeten 1990] J. C. M. Baeten and W. P. Weijland. Process algebra. Cambridge University Press, 1990. 19

[Bagnara 2007] Roberto Bagnara, Patricia M. Hill, Andrea Pescetti and Enea Zaffanella. *On the Design of Generic Static Analyzers for Modern Imperative Languages.* CoRR, vol. abs/cs/0703116, 2007. 131

[Benveniste 1991] Albert Benveniste, Paul Le Guernic and Christian Jacquemot. *Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics*. Sci. Comput. Program., vol. 16, no. 2, pages 103–149, 1991. 22

[Bergstra 1985] J.A. Bergstra and J.W. Klop. *Algebra of Communicating Processes with Abstraction*. Theoretical Computer Science, vol. 37, no. 1, pages 77–121, 1985. 19

[Berry 1992] G. Berry and G. Gonthier. *The* ESTEREL *Synchronous Programming Language: Design, Semantics, Implementation*. Science of Computer Programming, vol. 19, no. 2, pages 87–152, 1992. 13, 22, 26, 35, 132, 145, 148

[Bistarelli 1997] Stefano Bistarelli, Ugo Montanari and Francesca Rossi. *Semiring-based constraint satisfaction and optimization*. J. ACM, vol. 44, no. 2, pages 201–236, 1997. 55

[Blanchet 2005] Bruno Blanchet. *Security Protocols: From Linear to Classical Logic by Abstract Interpretation*. Information Processing Letters, vol. 95, no. 5, 2005. 111

[Bodei 1998] Chiara Bodei, Pierpaolo Degano, Flemming Nielson and Hanne Riis Nielson. *Control Flow Analysis for the pi-calculus*. In CONCUR, pages 84–98, 1998. 131

[Boreale 1996] Michele Boreale and Rocco De Nicola. *A Symbolic Semantics for the pi-Calculus*. Information and Computation, vol. 126, 1996. 55

[Boreale 2000] M. Boreale. *Symbolic analysis of cryptographic protocols in the spi-calculus*, 2000. 105, 111

[Boreale 2001a] M. Boreale and M. Buscemi. *A framework for the analysis of security protocols*, 2001. An abstract appears in Proc. of WSDAAL 2001, Como, Italy. 105

[Boreale 2001b] Michele Boreale. *Symbolic Trace Analysis of Cryptographic Protocols*. In Proc. of ICALP'01. LNCS, 2001. 55

[Boreale 2006] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos and Gianluigi Zavattaro. *SCC: A Service Centered Calculus*. In Proc. of WS-FM, volume 4184 of *LNCS*, pages 38–57. Springer, 2006. 114

[Borger 2001] E. Borger, E. Gradel and Y. Gurevich. The classical decision problem. Springer, 2001. 67

[Bortolussi 2008] Luca Bortolussi and Alberto Policriti. *Modeling Biological Systems in Stochastic Concurrent Constraint Programming*. Constraints, vol. 13, no. 1-2, 2008. 11

[Buchi 1962] J. R. Buchi. *On a decision method in restricted second order arithmetic*. In Proc. of Int. Conf. on Logic, Methodology, and Philosophy of Science, 1962. 14, 67

[Buscemi 2007] Maria Grazia Buscemi and Ugo Montanari. *CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements*. In In Proc. of ESOP, 2007. 11, 12, 37, 55, 130

[Buscemi 2008] Maria Grazia Buscemi and Ugo Montanari. *Open Bisimulation for the Concurrent Constraint Pi-Calculus*. In ESOP, volume 4960 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2008. 12, 55

[Busi 2003] Nadia Busi, Maurizio Gabbrielli and Gianluigi Zavattaro. *Replication vs. Recursive Definitions in Channel Based Calculi.* In ICALP, volume 2719 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2003. 86

[Busi 2004] Nadia Busi, Maurizio Gabbrielli and Gianluigi Zavattaro. *Comparing Recursion, Replication, and Iteration in Process Calculi.* In ICALP, volume 3142 of *Lecture Notes in Computer Science*, pages 307–319. Springer, 2004. 86

[Caspi 1999] Paul Caspi and Marc Pouzet. *Lucid Synchrone: une extension fonctionnelle de Lustre.* In Journées Francophones des Langages Applicatifs (JFLA), Morzine-Avoriaz, February 1999. INRIA. 22

[Codish 1994] M. Codish and B. Demoen. *Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop.* In Proc. of SAS'94, pages 281–296. Springer-Verlag, LNCS 864, 1994. 144

[Codish 1999] M. Codish, H. Søndergaard and P. Stuckey. *Sharing and groundness dependencies in logic programs.* ACM Trans. Program. Lang. Syst., vol. 21, no. 5, 1999. 132, 148

[Comini 1999] Marco Comini, Giorgio Levi, Maria Chiara Meo and Giuliana Vitiello. *Abstract Diagnosis.* J. Log. Program., vol. 39, no. 1-3, pages 43–93, 1999. 151

[Comini 2003] Marco Comini, Roberta Gori, Giorgio Levi and Paolo Volpe. *Abstract interpretation based verification of logic programs.* Sci. Comput. Program., vol. 49, no. 1-3, pages 89–123, 2003. 132

[Coppo 2008] Mario Coppo and Mariangiola Dezani-Ciancaglini. *Structured Communications with Concurrent Constraints.* In Proc. of TGC'08, LNCS. Springer, 2008. 130

[Cortesi 1997] Agostino Cortesi, Gilberto Filé, Francesco Ranzato, Roberto Giacobazzi and Catuscia Palamidessi. *Complementation in abstract interpretation.* ACM Trans. Program. Lang. Syst., vol. 19, no. 1, 1997. 96

[Cousot 1977] P. Cousot and R. Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In Proc. of Fourth ACM Symp. on Principles of Programming Languages, pages 238–252, 1977. 16, 131

[Cousot 1979] Patrick Cousot and Radhia Cousot. *Systematic Design of Program Analysis Frameworks.* In POPL, pages 269–282, 1979. 10, 131

[Cousot 1992] Patrick Cousot and Radhia Cousot. *Abstract Interpretation and Application to Logic Programs.* J. Log. Program., vol. 13, no. 2&3, pages 103–179, 1992. 132, 139, 142, 144, 148

[Crazzolara 2001] Federico Crazzolara and Glynn Winskel. *Petri nets in cryptographic protocols.* In IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium, USA, 2001. IEEE CS. 105, 111

[de Boer 1995a] F. de Boer, A. Di Pierro and C. Palamidessi. *Nondeterminism and infinite computations in constraint programming.* Theoretical Computer Science, vol. 151, no. 1, pages 37–78, 13 November 1995. 150

[de Boer 1995b] Frank S. de Boer, Alessandra Di Pierro and Catuscia Palamidessi. *Nondeterminism and Infinite Computations in Constraint Programming*. Theor. Comput. Sci., vol. 151, no. 1, pages 37–78, 1995. 103, 132, 134, 138, 139

[de Boer 1997] F. S. de Boer, M. Gabbrielli, E. Marchiori and C. Palamidessi. *Proving Concurrent constraint Programs Correct*. ACM Transactions on Programming Languages and Systems, vol. 19, no. 5, 1997. 12, 54, 57, 66, 96, 150

[Degtyarev 2002] Anatoli Degtyarev, Michael Fisher and Alexei Lisitsa. *Equality and Monodic First-Order Temporal Logic*. Studia Logica, vol. 72, no. 2, 2002. 86, 150

[Denning 1981] Dorothy E. Denning and Giovanni Maria Sacco. *Timestamps in key distribution protocols*. Commun. ACM, vol. 24, no. 8, 1981. 147

[Díaz 1998] Juan Francisco Díaz, Camilo Rueda and Frank D. Valencia. *Pi+- Calculus: A Calculus for Concurrent Processes with Constraints*. CLEI Electron. J., vol. 1, no. 2, 1998. 37, 130

[Dolev 1983] D. Dolev and A. C. Yao. *On the security of public key protocols*. IEEE Transactions on Information Theory, vol. 29, no. 12, 1983. 106, 112

[Fages 1998] Francois Fages, Sylvain Soliman and Victor Vianu. *Expressiveness and Complexity of Concurrent Constraint Programming: a Finite Model Theoretic Approach*. Rapport technique LIENS research report Liens-98-14, 1998. 26

[Fages 2001] Francois Fages, Paul Ruet and Sylvain Soliman. *Linear Concurrent Constraint Programming: Operational and Phase Semantics*. Information and Computation, vol. 165, 2001. 12, 37, 57, 150

[Falaschi 1993] M. Falaschi, M. Gabbrielli, K. Marriott and C. Palamidessi. *Compositional Analysis for Concurrent Constraint Programming*. In Proc. of LICS'93, 1993. 141

[Falaschi 1997] M. Falaschi, M. Gabbrielli, K. Marriott and C. Palamidessi. *Confluence in Concurrent Constraint Programming*. Theoretical Computer Science, vol. 183, no. 2, pages 281–315, 1997. 117, 132, 139, 141, 150

[Falaschi 2007] M. Falaschi, C. Olarte, C. Palamidessi and F. Valencia. *Declarative Diagnosis of Temporal Concurrent Constraint Programs*. In Proc. of ICLP'07. Springer LNCS 4670, 2007. 16, 148, 151

[Falaschi 2009] Moreno Falaschi, Carlos Olarte and Catuscia Palamidessi. *A Framework for Abstract Interpretation of Universal Timed Concurrent Constraint Programs*. In Proc. of PPDP'09. ACM, 2009. To Appear. 16, 148

[Feret 2005] Jérôme Feret. *Abstract interpretation of mobile systems*. J. Log. Algebr. Program., vol. 63, no. 1, pages 59–130, 2005. 131, 148

[Fiore 2001] Marcelo Fiore and Martin Abadi. *Computing Symbolic Models for Verifying Cryptographic Protocols*. In Proc. of the 14th IEEE Workshop on Computer Security Foundations. IEEE CS, 2001. 33, 55, 106

[Fournet 2003] C. Fournet and M. Abadi. *Hiding names: Private authentication in the applied pi calculus*, 2003. 105, 111, 112

[Garoche 2007] P.-L. Garoche, M. Pantel and X. Thiroux. *Abstract Interpretation-based Static Safety for Actors.* Journal of Software, vol. 2, no. 3, pages 87–98, 2007. 131, 148

[Giesl 2007] Jürgen Giesl, Peter Schneider-Kamp, René Thiemann, Stephan Swiderski, Manh Thang Nguyen, Daniel De Schreye and Alexander Serebrenik. *Termination of Programs using Term Rewriting and SAT Solving.* In Deduction and Decision Procedures. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. 131

[Girard 1987] Jean-Yves Girard. *Linear Logic.* Theor. Comput. Sci., vol. 50, pages 1–102, 1987. 37, 111

[Gupta 1996a] Vineet Gupta, Radha Jagadeesan and Vijay Saraswat. *Hybrid cc, hybrid automata and program verification (extended abstract).* In Proc. of the DIMACS/SYCON workshop on Hybrid systems III : verification and control, USA, 1996. Springer-Verlag. 11

[Gupta 1996b] Vineet Gupta, Radha Jagadeesan and Vijay A. Saraswat. *Models for Concurrent Constraint Programming.* In CONCUR, volume 1119 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 1996. 132

[Halbwachs 1991] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. *The synchronous dataflow programming language Lustre.* Proceedings of the IEEE, vol. 79, no. 9, pages 1305–1320, September 1991. 22, 132, 145

[Hildebrandt 2009] Thomas Hildebrandt and Hugo A. Lopez. *Types for Secure Pattern Matching with Local Knowledge in Universal Concurrent Constraint Programming.* In Proc. of ICLP'09. Springer LNCS, 2009. 11, 33, 37, 111, 112, 132, 150

[Hoare 1985] C. A. R. Hoare. Communications sequential processes. Prentice-Hall, Englewood Cliffs (NJ), USA, 1985. 11, 19, 150

[Hodkinson 2000] Ian M. Hodkinson, Frank Wolter and Michael Zakharyaschev. *Decidable fragment of first-order temporal logics.* Ann. Pure Appl. Logic, vol. 106, no. 1-3, 2000. 67, 86

[Honda 1998] Kohei Honda, Vasco Thudichum Vasconcelos and Makoto Kubo. *Language Primitives and Type Discipline for Structured Communication-Based Programming.* In Proc. of ESOP, volume 1381 of *LNCS*. Springer, 1998. 10, 15, 113, 114, 115, 116, 117, 119, 129, 130, 151

[Hussak 2008] Walter Hussak. *Decidable Cases of First-order Temporal Logic with Functions.* Studia Logica, vol. 88, no. 2, pages 247–261, 2008. 86

[Jaffar 1987] Joxan Jaffar and Jean-Louis Lassez. *Constraint Logic Programming.* In Proc. of POPL 87, 1987. 21

[Lanese 2007] Ivan Lanese, Francisco Martins, Vasco Thudichum Vasconcelos and António Ravara. *Disciplining Orchestration and Conversation in Service-Oriented Computing.* In Proc. of SEFM, pages 305–314. IEEE Computer Society, 2007. 114

[Laneve 1992] Cosimo Laneve and Ugo Montanari. *Mobility in the CC-Paradigm.* In Mathematical Foundations of Computer Science, 1992. 12, 36

[Lapadula 2007] A. Lapadula, R. Pugliese and F. Tiezzi. *A calculus for orchestration of web services*. In Proc. of ESOP, volume 4421 of *LNCS*, pages 33–47. Springer, 2007. 114

[Lopez 2009] Hugo Lopez, Carlos Olarte and Jorge A. Pérez. *Towards a Unified Framework for Declarative Sessions*. In Proc. of PLACES'09, 2009. 17, 129

[Lowe 1996] Gavin Lowe. *Breaking and fixing the Needham-Schroeder public-key protocol using FDR*. In Proc. of TACAS'96. LNCS, 1996. 108, 111, 147

[Manna 1991] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems: Specification. Springer-Verlag, 1991. 9, 13, 14, 22, 23, 39, 57, 65, 130, 149

[Mendler 1995] N P Mendler, Prakash Panangaden, Philip J Scott and R A G Seely. *A Logical View of Concurrent Constraint Programming*. Nord. J. Comput., vol. 2, no. 2, pages 181–220, 1995. 48, 50, 57, 66, 132

[Merz 1992] Stephan Merz. *Decidability and incompleteness results for first-order temporal logics of linear time*. Journal of Applied Non-Classical Logics, vol. 2, no. 2, 1992. 67, 78, 80, 85, 86, 150

[Mesnard 2005] Frédéric Mesnard and Roberto Bagnara. *cTI: A constraint-based termination inference tool for ISO-Prolog*. TPLP, vol. 5, no. 1-2, pages 243–257, 2005. 131

[Milner 1989] R. Milner. Communication and concurrency. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24. 11, 19

[Milner 1992a] R. Milner. *A finite delay operator in Synchronous CCS*. Rapport technique CSR-116-82, University of Edinburgh, 1992. 86

[Milner 1992b] R. Milner, J. Parrow and D. Walker. *A Calculus of Mobile Processes, Parts I and II*. Journal of Information and Computation, vol. 100, September 1992. 11, 19, 20, 27, 32, 80, 85, 86, 148, 150

[Milner 1992c] Robin Milner. *The Polyadic Pi-calculus (Abstract)*. In CONCUR, volume 630 of *Lecture Notes in Computer Science*. Springer, 1992. 28

[Milner 1999] Robin Milner. Communicating and mobile systems: the pi-calculus. Cambridge University Press, 1999. 11, 19, 20, 86

[Minsky 1967] Marvin L. Minsky. Computation: finite and infinite machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967. 9, 68

[Needham 1978] Roger M. Needham and Michael D. Schroeder. *Using encryption for authentication in large networks of computers*. Commun. ACM, vol. 21, no. 12, 1978. 107, 111, 131, 146

[Nielsen 2002a] M. Nielsen, C. Palamidessi and F.D. Valencia. *Temporal Concurrent Constraint Programming: Denotation, Logic and Applications*. Nordic Journal of Computing, vol. 9, no. 1, 2002. 11, 12, 14, 15, 25, 26, 28, 57, 66, 89, 93, 101, 103, 117, 132, 133, 134, 135, 138, 139, 150

[Nielsen 2002b] Mogens Nielsen, Catuscia Palamidessi and Frank D. Valencia. *On the expressive power of temporal concurrent constraint programming languages*. In PPDP, pages 156–167. ACM, 2002. 48, 50, 86

[Olarte 2007a] C. Olarte. *A Process Calculus for Universal Concurrent Constraint Programming: Semantics, Logic and Application.* In Newsletter of the ALP, Vol. 20 n. 3/4, December 2007. 17

[Olarte 2007b] Carlos Olarte, Catuscia Palamidessi and Frank Valencia. *Universal Timed Concurrent Constraint Programming.* In ICLP, pages 464–465, 2007. 17

[Olarte 2008a] C. Olarte, C. Rueda and F. Valencia. *Concurrent Constraint Programming: Calculi, Languages and Emerging Applications.* In Newsletter of the ALP, Vol. 21 n. 2-3, August 2008. 11, 17, 132

[Olarte 2008b] Carlos Olarte and Frank D. Valencia. *The Expressivity of Universal Timed CCP: Undecidability of Monadic FLTL and Closure Operators for Security.* In Proc. of PPDP 08. ACM, 2008. 16, 85, 103, 111, 132

[Olarte 2008c] Carlos Olarte and Frank D. Valencia. *Universal Concurrent Constraint Programing: Symbolic Semantics and Applications to Security.* In Proc. of SAC 2008. ACM, 2008. 36, 55, 65, 111, 132

[Olarte 2009a] C. Olarte, C. Rueda and F. Valencia. Concurrent constraint calculi: a declarative paradigm for modeling music systems, pages 93–112. Delatour France / IRCAM-Centre Pompidou, 2009. 17

[Olarte 2009b] Carlos Olarte and Camilo Rueda. *A declarative language for dynamic multimedia interaction systems.* In Proc. of MCM. Springer-Verlag, 2009. 16, 129

[Palamidessi 2006] Catuscia Palamidessi, Vijay Saraswat, Frank Valencia and Bjorn Victor. *On the Expressiveness of Linearity vs Persistence in the Asychronous Pi-Calculus.* In Proc. of LICS'06. IEEE CS, 2006. 13, 66, 86

[Pesic 2006] Maja Pesic and Wil M. P. van der Aalst. *A Declarative Approach for Flexible Business Processes Management.* In BPM'06 Workshops, volume 4103 of *LNCS*, pages 169–180. Springer, 2006. 114, 123

[Plotkin 1981] G. Plotkin. *A Structural Approach to Operational Semantics.* Rapport technique FN-19, DAIMI, University of Aarhus, 1981. 29

[Pnueli 1977] A. Pnueli. *The temporal logic of programs.* In Proc. of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77), pages 46–57. IEEE, IEEE Computer Society Press, 1977. 22

[Rueda 2004] C. Rueda and F. Valencia. *On validity in modelization of musical problems by CCP.* Soft Comput., vol. 8, no. 9, 2004. 11

[Sangiorgi 1992] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992. 80, 81, 84

[Sangiorgi 1996] Davide Sangiorgi. *A Theory of Bisimulation for the pi-Calculus.* Acta Inf., vol. 33, no. 1, pages 69–97, 1996. 55

[Sangiorgi 1998] D. Sangiorgi. *Interpreting functions as pi-calculus processes: a tutorial.* Rapport technique RR-3470, INRIA Sophia Antipolis, 1998. 80, 81, 85, 86

[Sangiorgi 2001] D. Sangiorgi and D. Walker. The π-calculus: a Theory of Mobile Processes. Cambridge University Press, 2001. 11, 19, 20, 27, 32, 86, 148, 150

[Saraswat 1991] Vijay A. Saraswat, Martin Rinard and Prakash Panangaden. *The semantic foundations of concurrent constraint programming.* In POPL '91. ACM Press, 1991. 12, 21, 103, 133, 134, 140, 143

[Saraswat 1992] Vijay Saraswat and Patrick Lincoln. *Higher-order Linear Concurrent Constraint Programming.* Rapport technique, 1992. 12, 87, 150

[Saraswat 1993] Vijay A. Saraswat. Concurrent constraint programming. MIT Press, 1993. 9, 11, 12, 21, 25, 26, 36, 37, 57, 133, 134, 149

[Saraswat 1994] Vijay Saraswat, Radha Jagadeesan and Vineet Gupta. *Foundations of Timed Concurrent Constraint Programming.* In Proc. of LICS'94. IEEE CS, 1994. 9, 11, 12, 13, 14, 15, 22, 25, 26, 27, 34, 57, 66, 67, 86, 89, 93, 95, 103, 132, 135, 138, 139, 145, 148, 149

[Scott 1982] Dana S. Scott. *Domains for Denotational Semantics.* In ICALP, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer, 1982. 12, 55, 89, 91, 132, 133

[Shapiro 1989] Ehud Shapiro. *The family of concurrent logic programming languages.* ACM Comput. Surv., vol. 21, no. 3, 1989. 21

[Smolka 1994] Gert Smolka. *A Foundation for Higher-order Concurrent Constraint Programming.* In In Proc. of CCL 94, 1994. 25, 26

[Song 2001] Dawn Xiaodong Song, Sergey Berezin and Adrian Perrig. *Athena: A Novel Approach to Efficient Automatic Security Protocol Analysis.* Journal of Computer Security, vol. 9, no. 1/2, pages 47–74, 2001. 146

[Szalas 1988] Andrzej Szalas and Leszek Holenderski. *Incompleteness of First-Order Temporal Logic with Until.* Theor. Comput. Sci., vol. 57, 1988. 67, 86

[Tini 1999] Simone Tini. *On The Expressiveness of Timed Concurrent Constraint Programming.* Electr. Notes Theor. Comput. Sci., vol. 27, 1999. 67, 132, 145, 148

[Ueda 2006] Kazunori Ueda, Norio Kato, Koji Hara and Ken Mizuno. *LMNtal as a Unifying Declarative Language: Live Demonstration.* In Proc. of ICLP 06. LNCS, 2006. 38

[Valencia 2005] Frank D. Valencia. *Decidability of infinite-state timed CCP processes and first-order LTL.* Theor. Comput. Sci., vol. 330, no. 3, 2005. 14, 27, 66, 67, 85, 86, 150

[Victor 1998] Björn Victor and Joachim Parrow. *Concurrent Constraints in the Fusion Calculus.* In Proc. of ICALP, volume 1443 of *LNCS*, pages 455–469. Springer, 1998. 130

[Wischik 2005] Lucian Wischik and Philippa Gardner. *Explicit fusions.* Theor. Comput. Sci., vol. 340, no. 3, pages 606–630, 2005. 12, 37

[Zaffanella 1997] E. Zaffanella, R. Giacobazzi and G. Levi. *Abstracting Synchronization in Concurrent Constraint Programming.* Journal of Functional and Logic Programming, vol. 1997, no. 6, 1997. 132, 139, 141, 142

Appendix A

Incompleteness of Monadic FLTL: Alternative Proof

In this appendix we present a more direct proof of the undecidability of monadic FLTL without equality nor function symbols. This proof relies solely on arguments of logic and not on the underlying theory of `utcc` as the one we presented in Chapter 6. Basically we prove that the FLTL formula corresponding to the process modeling the Minsky machine M faithfully captures the behavior of M. Then, we effectively build a formula that is valid if and only if M never halts. We refer the reader to the Chapter 6 for further discussions regarding this result.

A.1 Encoding Minsky Machines

Counters. The formulae modeling the two counters c_0 and c_1 are obtained by replacing the subindex n by 0 and 1 respectively in the formulae $\mathcal{F}zero_n$ and $\mathcal{F}not\text{-}zero_n$. Roughly speaking, the formula $\mathcal{F}zero_n$ models the state $c_n = 0$ and $\mathcal{F}not\text{-}zero_n$ the state $c_n = k$ for $k > 0$.

State Zero: Once $\texttt{zero}_n$ holds, $\texttt{isz}_n$ must also hold. We can then use the propositional variable $\texttt{isz}_n$ to test if the counter is zero.

If the current instruction does not modify the value of c_n, $\texttt{idle}_n$ must hold (due to the formula $\mathcal{F}$ins) and then, the formula $\circ\,\texttt{zero}_n$ must also hold. This way, we model the fact that the counter remains in zero.

State Not-Zero: When an increment instruction is executed, $\texttt{inc}_n$ holds (due to the formula $\mathcal{F}$ins), and so does a formula of the form $H = \circ\exists a.(not\text{-}zero_n(a) \wedge \Box(\texttt{out}(a) \Rightarrow F))$. In H, F is $\texttt{zero}_n$ if $c_n = 0$ (see $\mathcal{F}zero_n$) and $not\text{-}zero_n(x)$ otherwise (see $\mathcal{F}not\text{-}zero_n$). Intuitively, F represents the state immediately before the last increment instruction took place. This way, when a decrement instruction is performed, $\texttt{out}(a)$ holds and so does F.

Consider now $\mathcal{F}not\text{-}zero_n$ which is of the form $\forall x.not\text{-}zero_n(x) \Rightarrow G$. As we explained before, a formula of the form $H = \circ\exists a.(not\text{-}zero_n(a))$ holds when an increment instruction is performed. Using H in conjunction with $\mathcal{F}not\text{-}zero_n$ we obtain an instantiation of the form $\exists a.(G[a/x])$ that represents the state $c_n = k+1$. Notice that when $\exists a.(G[a/x])$ holds, $\texttt{isz}_n$ must not hold. Furthermore, similarly to the state zero, if the counter is not modified by the current instruction ($\texttt{idle}_n$ holds), $\circ not\text{-}zero_n(a)$ must hold and then, the counter takes the same value in the next time interval.

Instructions. For the set of instruction $(l_1, L_1); \ldots; (l_m, L_m)$ we assume a set of variables $l_1, \ldots, l_m$. If the predicate $\texttt{out}(l_i)$ holds in a state, it means that the instruction l_i is executed. In the case of a halt instruction $(l_i, \texttt{HALT})$, `halt` holds. For an increment or a decrement instruction $\neg\,\texttt{halt}$ holds.

The formula representing an increment operations $(l_i : \texttt{INC}(c_n, l_j))$ assures that $\texttt{inc}_n$ holds. It also guarantees that $\texttt{idle}_{1-n}$ holds while $\texttt{idle}_n$ does not.

Counters

$$\begin{array}{lcl}
\mathcal{F}zero_n & = & \mathtt{zero}_n \Rightarrow \left(\begin{array}{l} \mathtt{inc}_n \Rightarrow \circ\exists a.(\ not\text{-}zero_n(a) \wedge \\ \qquad\qquad \Box(\mathtt{out}(a) \Rightarrow \mathtt{zero}_n)) \\ \wedge\, \mathtt{idle}_n \Rightarrow \circ\, \mathtt{zero}_n \\ \wedge\, \mathtt{isz}_n \end{array} \right) \\
\mathcal{F}not\text{-}zero_n & = & \forall x.not\text{-}zero_n(x) \Rightarrow \\
 & & \left(\begin{array}{l} \mathtt{inc}_n \Rightarrow \circ\exists b.(\ not\text{-}zero_n(b) \wedge \\ \qquad\qquad \Box(\mathtt{out}(b) \Rightarrow not\text{-}zero_n(x))) \\ \wedge\, \mathtt{dec}_n \Rightarrow \circ\, \mathtt{out}(x) \\ \wedge\, \mathtt{idle}_n \Rightarrow \circ not\text{-}zero(x) \\ \wedge \neg\, \mathtt{isz}_n \end{array} \right)
\end{array}$$

Instructions

$$\begin{array}{lcl}
\mathcal{F}\mathrm{ins} & = & \bigwedge_{1 \le i \le m} \mathtt{out}(l_i) \Rightarrow [\![l_i : L_i]\!]_I \text{ where} \\
[\![l_i : \mathtt{HALT}]\!]_I & = & \mathtt{halt} \\
[\![l_i : \mathtt{INC}(c_n, l_j)]\!]_I & = & \neg\, \mathtt{halt} \wedge \mathtt{inc} \wedge \neg\, \mathtt{idle}_n \wedge \mathtt{idle}_{1-n} \wedge \circ\, \mathtt{out}(l_j) \\
[\![l_i : \mathtt{DECJ}(c_n, l_j, l_k)]\!]_I & = & \mathtt{isz} \Rightarrow (\mathtt{idle}_n \wedge \circ\, \mathtt{out}(l_j)) \wedge \\
 & & \neg\, \mathtt{isz} \Rightarrow (\neg\, \mathtt{idle}_n \wedge \mathtt{dec}_n \wedge \circ\, \mathtt{out}(l_k)) \wedge \\
 & & \mathtt{idle}_{1-n} \wedge \neg\, \mathtt{halt}
\end{array}$$

Figure A.1: Representation of a Minsky machine with instructions $(l_1 : L_1); ...; (l_m : L_m)$. The subindex $n \in \{0, 1\}$.

Finally, the formula representing a decrement instruction of the form $(l_i : \mathtt{DECJ}(c_n, l_j, l_k))$ tests if the counter c_n is zero. If this is the case, then it activates in the next time interval the instruction l_j. If $\mathtt{isz}_n$ does not hold, i.e. $c_n > 0$, $\mathtt{dec}_n$ must hold and the instruction l_k is activated in the next time unit.

The following definition introduces the formula $\mathcal{F}_M$ that simulates the behavior of a Minsky machine M.

Definition A.1.1 (Encoding of a Minsky Machine)**.** *Let M be a Minsky machine with instructions $(l_1 : L1), ..., (l_n : L_m)$. The encoding $[\![M]\!]$ is defined as the formula*

$$\mathcal{F}_M = \quad \Box(\mathcal{F}zero_0 \wedge \mathcal{F}not\text{-}zero_0 \wedge \mathcal{F}zero_1 \wedge \mathcal{F}not\text{-}zero_1 \wedge \mathcal{F}\mathrm{ins})$$

where $\mathcal{F}zero_0, \mathcal{F}not\text{-}zero_0, \mathcal{F}zero_1$ and $\mathcal{F}not\text{-}zero_1$ are obtained by replacing the sub-index n in the Equations in Figure A.1.

A.2 Encoding of Numbers and Configurations

To show that the formula $\mathcal{F}_M$ above faithfully describes the behavior of the machine M, we shall give first a suitable representation of numbers and configurations of M. This shall ease the forthcoming proofs.

As hinted at above, when an increment operations is performed, a formula of the form $H = \circ\exists a.(not\text{-}zero_n(a) \wedge \Box(\mathtt{out}(a) \Rightarrow F))$ must hold, where F represents the state immediately before the last increment instruction took place. Recall also that a decrement operation causes that $\mathtt{out}(a)$ holds and so does F

We can then represent the state $c_n = k$, for $k > 0$, as a formula of the form $\exists.a_1, .., a_k(F_1 \wedge ... \wedge F_k \wedge not\text{-}zero_n(a_k))$ where F_1 is of the form $\Box\, \mathtt{out}(a_1) \Rightarrow \mathtt{zero}_n$ and for $1 < i \le k$, $F_i = \Box\, \mathtt{out}(a_i) \Rightarrow not\text{-}zero_n(a_{i-1})$. More precisely,

Definition A.2.1 (Representation of Numbers)**.** *The FLTL formula representing the state $c_n = k$, notation $[\![c_n = k]\!]_N$, is defined as follows:*

$$\begin{array}{lcl}
[\![c_n = 0]\!]_N & = & \mathtt{zero}_n \\
[\![c_n = 1]\!]_N & = & \exists.a_1(\Box\,\mathtt{out}(a_1) \Rightarrow \mathtt{zero}_n \wedge \\
 & & \quad \mathit{not\text{-}zero}_n(a_1)) \\
\ldots & & \\
[\![c_n = k]\!]_N & = & \exists.a_1, a_2, ..., a_k(\ \Box\,\mathtt{out}(a_1) \Rightarrow \mathtt{zero}_n \wedge \\
 & & \quad \Box\,\mathtt{out}(a_2) \Rightarrow \mathit{not\text{-}zero}_n(a_1) \wedge \\
 & & \quad \ldots \\
 & & \quad \Box\,\mathtt{out}(a_k) \Rightarrow \mathit{not\text{-}zero}_n(a_{k-1}) \wedge \\
 & & \quad \mathit{not\text{-}zero}_n(a_k))
\end{array}$$

Using the previous definition of numbers, we define next the FLTL formula representing a configuration of a Minsky machine.

Definition A.2.2 (Encoding of Configurations)**.** *Let M be a Minsky machine with instructions $(l_1; L_1), ..., (l_m; L_m)$. Let $[\![\cdot]\!]_N$ be as in Definition A.2.1 and $\mathcal{F}_M$ be as in Definition A.1.1. The encoding $[\![\cdot]\!]_C$ of a configuration of M is defined as*

$$[\![(l_i, v_0, v_1)]\!]_C \quad = \quad \mathcal{F}_M \wedge [\![c_0 = v_0]\!]_N \wedge [\![c_1 = v_1]\!]_N \wedge \mathtt{out}(l_i)$$

A.3 Monadic FLTL is Undecidable

We shall use the above construction to exhibit a formula that is valid if and only if the machine M loops (i.e., it never halts). This shall allow us to show that this fragment of FLTL is incomplete, i.e., its set of tautologies is not recursively enumerable.

We start by proving that computations of M are faithfully described by the formula $\mathcal{F}_M$ in Definition A.1.1.

Lemma A.3.1 (Soundness of the Encoding)**.** *Let M be a Minsky machine with instructions $(l_1; L_1), ..., (l_m; L_m)$, $[\![\cdot]\!]_C$ be as in Definition A.2.2 and (l_i, v_0, v_1) be a configuration of M.*

$$\textit{If } (l_i, v_0, v_1) \longrightarrow_M (l_i', v_0', v_1') \textit{ then } [\![(l_i, v_0, v_1)]\!]_C \models \neg\,\mathtt{halt} \wedge \circ [\![(l_i', v_0', v_1')]\!]_C$$

Furthermore, if $(l_i, v_0, v_1) \not\longrightarrow_M$, i.e., l_i is a `HALT` *instruction, then it holds $[\![(l_i, v_0, v_1)]\!]_C \models$* `halt`*.*

Proof. First assume that $(l_i, v_0, v_1) \not\longrightarrow_M$. Then $(l_i : L_i)$ is a `HALT` instruction and it is easy to see that $[\![(l_i, v_0, v_1)]\!]_C \models \mathtt{halt}$ for any v_0, v_1.

Assume now that $(l_i, v_0, v_1) \longrightarrow_M (l_i', v_0', v_1')$. Then, $(l_i : L_i)$ must be an increment or a decrement instruction. It is easy to see that for both cases $[\![(l_i, v_0, v_1)]\!]_C \models \neg\,\mathtt{halt}$. Now we shall prove that $[\![(l_i, v_0, v_1)]\!]_C \models \circ [\![(l_i', v_0', v_1')]\!]_C$ for both kind of instructions.

- Assume that $(l_i : L_i)$ is of the form $(li : \mathtt{INC}(c_n, l_j))$. It must be the case that $l_i' = l_j$, $v_n' = v_n + 1$ and $v_{1-n}' = v_{1-n}$. Let $F = \mathtt{out}(l_i) \Rightarrow [\![(l_i : L_i)]\!]_I$ be the subformula in $[\![(l_i, v_0, v_1)]\!]_C$ representing the encoding of the instruction l_i. We know that $[\![(l_i, v_0, v_1)]\!]_c \models \mathtt{out}(l_i)$ and we can derive the following:

  $$F \wedge \mathtt{out}(l_i) \models \mathtt{inc} \wedge \neg\,\mathtt{idle}_n \wedge \mathtt{idle}_{1-n} \wedge \circ\,\mathtt{out}(l_j)$$

  For the counter c_{1-n}, it is easy to see that

  $$\mathcal{F}_M \wedge \mathtt{idle}_{1-n} \wedge [\![c_{1-n} = v_{1-n}]\!]_N \models \circ [\![c_{1-n} = v_{1-n}]\!]_N$$

For the counter c_n we consider two cases: $v_n = 0$ and $v_n > 0$. For $v_n = 0$ we can derive the following

$$\begin{array}{rcl} \mathcal{F}_M \wedge \mathtt{inc}_n \wedge [\![c_n = 0]\!]_N & \models & \circ\exists.a_1(\mathit{not\text{-}zero}(a_1) \wedge \\ & & \qquad \Box\,\mathtt{out}(a_1) \Rightarrow \mathtt{zero}_n) \\ & & \equiv \circ[\![c_n = 1]\!]_N \end{array}$$

If $v_n = k$ for $k > 0$, then we have

$$\begin{array}{rcl} \mathcal{F}_M \wedge \mathtt{inc}_n \wedge [\![c_n = k]\!]_N & \models & \circ\exists.a_1, ..., a_k, a_{k+1}(\\ & & \qquad \Box\,\mathtt{out}(a_1) \Rightarrow \mathtt{zero}_n \wedge \\ & & \qquad \ldots \\ & & \qquad \Box(\mathtt{out}(a_{k+1}) \Rightarrow \mathit{not\text{-}zero}_n(a_k)) \wedge \\ & & \qquad \mathit{not\text{-}zero}_n(a_{k+1})) \\ & & \equiv \circ[\![c_n = k+1]\!]_N \end{array}$$

We then conclude $[\![(l_i, v_0, v_1)]\!]_C \models \circ[\![(l_i', v_0', v_1')]\!]_C$.

- Assume now that $(l_i : L_i)$ is of the form $(l_i : \mathtt{DECJ}(c_n, l_j, l_k))$. We must consider two cases.
  - If $c_n = 0$ then it must be the case that $v_n' = v_n$ and $l_i' = l_j$. We can derive the following

$$\begin{array}{rcl} \mathcal{F}_M \wedge [\![c_n = 0]\!]_N & \models & \mathtt{isz}_n \wedge \mathtt{idle}_n \wedge \mathtt{idle}_{1-n} \wedge \circ(\mathtt{out}(l_j) \wedge \mathtt{zero}_n) \\ & \models & \circ[\![c_n = 0]\!]_N \end{array}$$

  - If $c_n = k$ for some $k > 0$, then we derive

$$\mathcal{F}_M \wedge [\![c_n = k]\!]_N \quad \models \quad \neg\,\mathtt{isz}_n \wedge \mathtt{dec}_n \wedge \mathtt{idle}_{1-n} \wedge \circ\,\mathtt{out}(l_k)$$

    Let $F' = \mathcal{F}_M \wedge [\![c_n = k]\!]_N \wedge \mathtt{dec}_n$. We derive the following

$$\begin{array}{rcl} F' & \models & \mathcal{F}_M \wedge \exists.a_1, ..., a_k(\ \Box\,\mathtt{out}(a_1) \Rightarrow \mathtt{zero}_n \wedge \\ & & \qquad \ldots \\ & & \qquad \Box\,\mathtt{out}(a_k) \Rightarrow \mathit{not\text{-}zero}_n(a_{k-1}) \\ & & \qquad \wedge \mathit{not\text{-}zero}_n(a_k) \wedge \circ\,\mathtt{out}(a_k)) \\ & \models & \circ\exists.a_1, ..., a_k(\Box\,\mathtt{out}(a_1) \Rightarrow \mathtt{zero}_n \wedge \\ & & \qquad \ldots \\ & & \qquad \Box\,\mathtt{out}(a_{k-1}) \Rightarrow \mathit{not\text{-}zero}_n(a_{k-2}) \\ & & \qquad \wedge \mathit{not\text{-}zero}_n(a_{k-1}) \\ & & \equiv \circ[\![c_n = k-1]\!]_N \end{array}$$

□

Using the previous lemma, we can show that a machine M produces an infinite run if and only if the formula $\mathcal{F}_M \Rightarrow \Box\neg\,\mathtt{halt}$ is valid.

Lemma A.3.2. *A Minsky machine M loops (i.e. it never halts) if and only if*

$$[\![(l_1, 0, 0)]\!]_C \models \Box\neg\,\mathtt{halt}$$

Proof. Let $v_0^1 = v_1^1 = 0$ and $l^1 = l_1$.

($\Rightarrow$) Assume that M produces an infinite run

$$(l^1, v_0^1, v_1^1) \longrightarrow_M (l^2, v_0^2, v_1^2) \ldots \longrightarrow_M (l^n, v_0^n, v_1^n) \longrightarrow_M \ldots$$

where for all $i \geq 1$, l^i is not a `HALT` instruction. From Lemma A.3.1 we know that for all $i \geq 1$, $[\![(l^i, v_0^i, v_1^i)]\!]_C \models \circ[\![(l^{i+1}, v_0^{i+1}, v_1^{i+1})]\!]_C$ and also that $[\![(l^i, v_0^i, v_1^i)]\!]_C \models \neg\, \mathtt{halt}$. Therefore, for $i \geq 0$, it holds

$$[\![(l^1, v_0^1, v_1^1)]\!]_C \models \circ^i(\neg\, \mathtt{halt})$$

where $\circ^0(F) \equiv F$ for any formula F. We then conclude $[\![l_1, 0, 0]\!]_C \models \Box\neg\, \mathtt{halt}$.

($\Leftarrow$) We proceed by contradiction. Assume that $[\![l_1, 0, 0]\!]_C \models \Box\neg\, \mathtt{halt}$ and there exists $n \geq 1$ such that the machine produces a run

$$(l^1, v_0^1, v_1^1) \longrightarrow_M (l^2, v_0^2, v_1^2) \ldots \longrightarrow_M (l^n, v_0^n, v_1^n) \not\longrightarrow_M$$

i.e., l^n is a `HALT` instruction. By Lemma A.3.1 we know that

$$[\![(l^1, v_0^1, v_1^1)]\!]_C \models \circ^{n-1}[\![(l^n, v_0^n, v_1^n)]\!]_C \models \circ^{n-1}\, \mathtt{halt} \models \Diamond\, \mathtt{halt}$$

thus a contradiction.

□

Since the set of looping Minsky machines (i.e. the complement of the halting problem) is not recursively enumerable, a finitistic axiomatization of monadic FLTL without equality nor function symbols would yield a non-recursively enumerable set of tautologies.

Theorem A.3.1 (Incompleteness)**.** *There is not a sound and complete finitistic axiomatization for monadic FLTL without equality nor function symbols.*

Proof. Directly from Lemma A.3.2. □

Index