Temporal Concurrent Constraint Programming
Frank D. Valencia Posso

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark
Temporal Concurrent Constraint Programming

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
PhD Degree

by
Frank D. Valencia Posso
November, 2002
In memory of my two mothers: Alicia and Inés
Abstract

Concurrent constraint programming (ccp) is a formalism for concurrency in which agents interact with one another by telling (adding) and asking (reading) information in a shared medium. Temporal ccp extends ccp by allowing agents to be constrained by time conditions. This dissertation studies temporal ccp as a model of concurrency for discrete-timed systems. The study is conducted by developing a process calculus called \( \text{nctc} \).

The \( \text{nctc} \) calculus generalizes the tcc model, the latter being a temporal ccp model for deterministic and synchronous timed reactive systems. The calculus is built upon few basic ideas but it captures several aspects of timed systems. As tcc, \( \text{nctc} \) can model unit delays, time-outs, pre-emption and synchrony. Additionally, it can model unbounded but finite delays, bounded eventuality, asynchrony and nondeterminism. The applicability of the calculus is illustrated with several interesting examples of discrete-time systems involving mutable data structures, robotic devices, multi-agent systems and music applications.

The calculus is provided with a denotational semantics that captures the reactive computations of processes in the presence of arbitrary environments. The denotation is proven to be fully abstract for a substantial fragment of the calculus. This dissertation identifies the exact technical problems (arising mainly from allowing nondeterminism, locality and time-outs in the calculus) that makes it impossible to obtain a fully abstract result for the full language of \( \text{nctc} \).

Also, the calculus is provided with a process logic for expressing temporal specifications of \( \text{nctc} \) processes. This dissertation introduces a (relatively) complete inference system to prove that a given \( \text{nctc} \) process satisfies a given formula in this logic.

The denotation, process logic and inference system presented in this dissertation significantly extend and strengthen similar developments for tcc and (untimed) ccp.

This dissertation addresses the decidability of various behavioral equivalences for the calculus and characterizes their corresponding induced congruences. The equivalences (and their associated congruences) are proven to be decidable for a significant fragment of the calculus. The decidability results involve a systematic translation of processes into finite state Büchi automata. To the author's best knowledge the study of decidability for ccp equivalences is original to this work.

Furthermore, this dissertation deepens the understanding of previous ccp work by establishing an expressive power hierarchy of several temporal ccp lan-

vii
guages which were proposed in the literature by other authors. These languages, represented in this dissertation as \textit{variants} of ntcc, differ in their way of defining infinite behavior (i.e., \textit{replication} or \textit{recursion}) and the scope of variables (i.e., \textit{static} or \textit{dynamic scope}). In particular, it is shown that (1) recursive procedures with parameters can be encoded into parameterless recursive procedures with dynamic scoping, and vice-versa; (2) replication can be encoded into parameterless recursive procedures with static scoping, and vice-versa; (3) the languages from (1) are \textit{strictly more expressive} than the languages from (2). (Thus, in this family of languages recursion is more expressive than replication and dynamic scope is more expressive than static scope.) Moreover, it is shown that behavioral equivalence is \textit{undecidable} for the languages from (1), but \textit{decidable} for the languages from (2). Interestingly, the undecidability result holds even if the process variables take values from a \textit{fixed finite domain} whilst the decidability holds for \textit{arbitrary domains}.

Both the expressive power hierarchy and decidability/undecidability results give a clear distinction among the various temporal ccp languages. Also, the distinction between static and dynamic scoping helps to clarify the situation in the (untimed) ccp family of languages. Moreover, the methods used in the decidability results may provide a framework to perform further systematic investigations of temporal ccp languages.
Acknowledgments

I owe much to Mogens Nielsen for his outstanding supervision; always guiding me in the right direction. Mogens gave me that chance and inspiration every PhD student needs to grow as a researcher. I am indebted to Catuscia Palamidessi for her insight into concurrency, her support, her warmth. Having met Catuscia was, to say the least, a very important event in my life. I am most grateful to Camilo Rueda for giving me constant advice and support during my PhD and even long before. Not only do I want to express my gratitude to Mogens, Catuscia and Camilo, but also my deepest admiration for their work, honesty and modesty. They are truly role models for me. I am very fortunate to have worked with them on the subject of this dissertation.

I am also indebted to many people with whom I have discussed my PhD work: Maurizio Gabbieli, Radha Jagadeesan, Dale Miller, Vineet Gupta, Vijay Saraswat and Prakash Panangaden. I have greatly benefited from their experience and constructive criticism.

Many thanks to Luca Aceto and Ugo Montanari for having accepted to evaluate this dissertation. Having them as examiners is certainly a great honor for me.

I would like to thank my fellow PhD students Daniele, Federico, Giuseppe, Jesus, Jiri, Marco, Mario, Oliver, Paulo, Pawel and Stefan Dantchev. Some of them proofread my work, some others had (usually constructive--) criticism about it. But above all I thank them for being my friends.

I would like to express my appreciation of the excellent working environment at BRICS, University of Aarhus. Special thanks goes to Janne Christensen, Karen Møller, Ulrich Kohlenbach, Uffe Engberg and Ingrid Larsen. I thank the Danish National Research Foundation for funding my PhD research at BRICS.

I would also like to express how fortunate I feel to have spent four years of my life in the friendly and beautiful city of Aarhus.

I am also grateful to my family as well as to Maribell Sacanamboy, Laura Feldt and Marina Bergstroem. Their love and support during my PhD were fundamental to me.

I dedicate this dissertation to the two most important persons in my life: Alicia Posso and Inés Posso; two women, my two mothers and this says it all.

Frank D. Valencia Posso,
Århus, November 2002.
# Contents

Abstract vii

Acknowledgments ix

1 Introduction 1
   1.1 Motivation ............................................. 1
      1.1.1 Problem Domain: Concurrency .................... 1
      1.1.2 The Problem Description: Reasoning about Concurrency 2
      1.1.3 Solution: Models of Concurrency ................ 3
      1.1.4 Model Extensions ................................. 4
      1.1.5 This Dissertation: A Temporal Extension of CCP ................ 5
   1.2 Background ............................................ 7
      1.2.1 Concurrent Constraint Programming ............... 7
      1.2.2 Reactive Concurrent Constraint Programming .... 10
      1.2.3 Process Calculi ................................... 11
   1.3 Character and Organization .......................... 20
      1.3.1 Nondeterminism and Asynchrony .................. 20
      1.3.2 Organization ..................................... 21
   1.4 Contributions ........................................ 23

2 Intuitive Description of ntcc 25
   2.1 Intuitive Description of Constraint Systems ........... 25
   2.2 Intuitive Description of Processes ..................... 26
      2.2.1 Communication: Telling and Asking Information .... 26
      2.2.2 Nondeterminism .................................. 27
      2.2.3 Parallel Composition ............................. 28
      2.2.4 Local Behavior ................................... 29
      2.2.5 Unit Delays and Time-Outs ........................ 29
      2.2.6 Asynchrony ....................................... 30
      2.2.7 Infinite Behavior ................................ 31
      2.2.8 Some Derived Forms ............................... 31
   2.3 Summary and Related Work ................................ 33

3 Operational Semantics 35
   3.1 Constraint Systems ..................................... 35
   3.2 Process Syntax and Notation ............................ 37
   3.3 Transitions ........................................... 38
3.3.1 Structural Congruence ................................. 39
3.3.2 Reduction Relations ................................. 41
3.3.3 Reduction Example ................................. 44
3.4 Basic Reduction Properties ............................ 46
3.5 Observable Behavior ................................. 49
   3.5.1 Interpreting Process Runs ...................... 49
   3.5.2 Summary of Observable Behavior and Equivalences . 50
3.6 Summary and Related Work .......................... 51

4 Denotational Semantics ................................. 53
   4.1 The Semantic Definitions .......................... 53
   4.2 Soundness .......................................... 55
   4.3 Completeness of the Denotation .................... 59
      4.3.1 The Locally Independent Fragment ............ 60
      4.3.2 The Proof of Completeness .................... 63
   4.4 Determinism and Monotonicity ...................... 67
   4.5 Summary and Related Work ........................ 70

5 Logic and Specification ................................. 71
   5.1 A Linear-Temporal Logic ........................... 71
   5.2 Specification ...................................... 73
      5.2.1 Proof System .................................. 73
      5.2.2 Derivation Example ............................ 75
      5.2.3 Soundness and Completeness of the Proof System . 76
   5.3 Summary and Related Work ........................ 79

6 Applications .......................................... 81
   6.1 Recursion ......................................... 81
      6.1.1 The Encoding .................................. 82
      6.1.2 Proof Principles for Recursion ............... 83
   6.2 Cell Example ...................................... 84
   6.3 RCX Robots: The Zigzagging Example ............. 87
   6.4 Multi-Agent Systems: The Pursuit Example ........ 87
   6.5 Musical Applications .............................. 90
      6.5.1 Controlled Improvisation ..................... 90
      6.5.2 Rhythm Patterns ............................... 92
   6.6 Summary .......................................... 93

7 Observable Behavior: Decidability and Congruence Issues 95
   7.1 Process Equivalences and Induced Congruences .... 96
   7.2 Characterization of Behavioral Equivalences ....... 98
      7.2.1 The Finite Case ................................ 99
      7.2.2 The General Case ............................. 103
   7.3 Decidability of Observable Equivalences ........... 108
      7.3.1 Infinite Number of States from Nondeterminism . 108
      7.3.2 Finite-State Representation for Restricted Nondeterminism 109
      7.3.3 Büchi Automata Representation ............... 113
7.3.4 Output Büchi Automata ........................................ 114
7.3.5 Input-Output Büchi Automata Representation .......... 115
7.4 Summary and Related Work ....................................... 116

8 Variations of the Calculus: The TCC Hierarchy 119
8.1 TCC Languages ...................................................... 120
  8.1.1 Replication ...................................................... 120
  8.1.2 Recursion ....................................................... 121
  8.1.3 Parameterless Recursion .................................... 123
  8.1.4 Summary of TCC Languages ............................... 125
8.2 The TCC Equivalences ............................................ 126
8.3 Undecidability Results ........................................... 126
  8.3.1 Undecidability over Finite-Domains .................... 127
  8.3.2 The Post’s Correspondence Problem Reduction ....... 127
  8.3.3 Undecidability Over Fixed Finite-Domains ........... 129
8.4 Decidability Results ............................................. 130
8.5 Classification of the TCC Languages ......................... 131
8.6 The Encodings ..................................................... 132
  8.6.1 Encoding rec_a → rep ...................................... 133
  8.6.2 Encoding rec_1 → rep ..................................... 133
  8.6.3 Encoding rep → rec_1 ...................................... 134
  8.6.4 Encoding rec_d → rec_p .................................... 135
  8.6.5 Encoding rec_p → rec_d .................................... 135
  8.6.6 Encoding rep → rec_a ...................................... 135
8.7 Correctness of the Encodings Using a Simulation Technique .... 136
  8.7.1 A Simulation Proof Technique ............................. 136
8.8 Summary and Related Work ..................................... 140

9 Concluding Remarks 143
  9.1 More on Related Work ........................................... 143
  9.2 Summary .......................................................... 145
  9.3 Future Directions ................................................ 145

A Proof of Correctness of the PCP Reduction 147
A.1 Properties of the PCP Reduction ................................. 147
A.2 Proof of Lemma 8.3.3 ............................................. 151

B Proof of Correctness of the Encodings 153
  B.1 Correctness of the encoding rep → rec_1 .................. 153
  B.2 Correctness of the encoding rec_d → rec_p ............... 154
  B.3 Correctness of [ ] : rec_a → rep ............................ 155
  B.4 Correctness of [ ] : rec_1 → rep ............................ 159
  B.5 Correctness of the encoding rec_p → rec_d ............... 160
  B.6 Correctness of [ ] : rep → rec_a ............................ 162

Bibliography 167

xiii
List of Tables

1.1 An operational semantics for a process calculus ............... 13
3.1 Operational Semantics of ntcc ................................. 42
4.1 Denotational Semantics of ntcc ................................. 56
5.1 Proof System for ntcc ........................................ 76
List of Figures

1.1 A ccp scenario .................................................. 9
8.1 The tcc hierarchy .............................................. 131
Chapter 1

Introduction

I believe that the right ideas to explain concurrency will only come from a dialectic between models from logic and mathematics and a proper distillation of a practical experience
— Robin Milner

This dissertation studies temporal concurrent constraint programming as a model of concurrency for systems of agents which interact with one another by posting constraints on both their shared variables and timed behavior. The thesis is that this model is suitable for describing and analyzing discrete-timed reactive systems.

1.1 Motivation

This section motivates the present work by casting it into a broader perspective: the development of models for concurrency.

1.1.1 Problem Domain: Concurrency

Concurrency is concerned with the fundamental aspects of systems consisting of multiple computing agents, usually called processes, that interact among each other. This covers a vast variety of systems which nowadays, due to technological advances such as the Internet, programmable robotic devices and mobile computing, most people can easily relate to. Some examples are:

- **Message-passing** communication based systems: Agents interact by exchanging messages. For instance, e-mail communication on the Internet, or robot point-to-point exchange of messages via infra-red communication.

- **Shared-Variables** communication based systems: Agents communicate by posting and reading information from a central location. For instance, reading and posting information on a server as in an Internet newsgroup. In the context of co-operative robotic devices, there can be a central control, usually a PC, on which the robots can post and read information (e.g., their relative positions).
• **Synchronous** systems: As opposed to *asynchronous* systems, in synchronous systems, agents need to synchronize with one another. In the Internet telephony services the caller and the callee's terminal need to synchronize to establish communication. In systems of mobile robotic devices, robots most certainly need to synchronize, e.g., to avoid bumping into each other. An example of asynchrony is SMS communication on mobile phones.

• **Reactive** systems: Involve systems that maintain an ongoing interaction with their environment. For instance, reservation systems and databases on the Internet. Co-operative robotic devices are typically programmed to react to their surroundings, e.g., going backwards whenever a touch sensor is pressed.

• **Timed** systems: Systems in which the agents are constrained by temporal requirements. For example, browser applications are constrained by timer-based exit conditions (i.e., *time-outs*) for the case in which a server cannot be contacted. E-mailer applications can be required to check for messages every $k$ time units. Also, robots can be programmed with timeouts (e.g., to wait for some signal) and with timed instructions (e.g., to go forward for 42 time-units).

• **Mobile** systems: Agents can change their communication links. This is the essence of mobile computing devices. For example, portable computers can connect to the Internet from different locations. Robotic devices also exhibit mobility since as they are on the move they may change their communication configuration. E.g., robots which could initially communicate with one another, may sometime later be too far away to continue to do so.

• **Secure** systems: Systems in which critical resources of some sort (e.g., secret information) must not be accessed, misused or modified by unwanted agents. Credit card usage on the Internet is now a common practice involving secure systems. To a more physical level, one now hears of robotic security systems [CHT99] which involve mobile devices that are strategically programmed to patrol, detect intruders and respond accordingly.

The above are but a few representatives of systems exhibiting concurrency, often referred to as *concurrent* systems. Furthermore, they can be combined to give rise to very complex concurrent systems; for example the Internet itself.

### 1.1.2 The Problem Description: Reasoning about Concurrency

The previous examples illustrate the practical relevance, complexity and ubiquity of concurrent systems. It is therefore crucial to be able to describe, analyze and, in general, reason about concurrent behavior. This reasoning must be precise and reliable. Consequently, it ought to be founded upon mathematical principles in the same way as the reasoning about the behavior of sequential
1.1. Motivation

programs is founded upon logic, domain theory and other mathematical disciplines.

Nevertheless, giving mathematical foundations to concurrent computation has become a serious challenge for computer science. Traditional mathematical models of (sequential) computation based on functions from inputs to outputs no longer apply. The crux is that concurrent computation, e.g., in a reactive system, is seldom expected to terminate, it involves constant interaction with the environment, and it is non-deterministic owing to unpredictable interactions among agents.

1.1.3 Solution: Models of Concurrency

Computer science has therefore taken up the task of developing models, conceptually different from those of sequential computation, for the precise understanding of the behavior of concurrent systems. Such models, as other scientific models of reality, are expected to satisfy the following criteria:

- They must be simple, i.e., based upon few basic principles.
- They must be expressive, i.e., capable of capturing interesting real-world situations.
- They must be formal, i.e., founded upon mathematical principles.
- They must provide techniques to allow reasoning about their particular focus.

In order to develop a model of concurrency one could suggest the following general strategy: Seize upon a few pervasive aspects of concurrency (e.g., synchronous communication), make them the focus of a model, and then submit the model to the above criteria. This strategy can be claimed to have been involved in the development of a mature collection of models for various aspects of concurrency. Some representatives of this collection are mentioned next.

Representative models for synchronous communication. Some of the most mature and well-known models of concurrency are process calculi like Milner’s CCS [Mil89], Hoare’s CSP [Hoa85], and ACP (developed by Bergstra and Klop [BK85] and also by Baeten [BW90]). The common focus of these models is synchronous communication.

Process calculi treat processes much like the λ-calculus treats computable functions. They provide a language in which the structure of terms represents the structure of processes together with an operational semantics to represent computational steps. For example, the term $P \parallel Q$, which is built from $P$ and $Q$ with the constructor $\parallel$, represents the process that results from the parallel execution of those represented by $P$ and $Q$. An operational semantics may dictate that if $P$ can evolve into $P'$ in a computational step $P'$ then $P \parallel Q$ can also evolve into $P' \parallel Q$ in a computational step.

An appealing feature of process calculi is their algebraic treatment of processes. The constructors are viewed as the operators of an algebraic theory whose
equations and inequalities among terms relate process behavior. For instance, the construct $\parallel$ can be viewed as a commutative operator, hence the equation $P \parallel Q \equiv Q \parallel P$ states that the behavior of the two parallel compositions are the same. Because of this algebraic emphasis, these calculi are often referred to as process algebras.

**A representative model for true-concurrency.** Another important model of concurrency is Petri Nets [Pet62]. The focus of Petri Nets is the simultaneous occurrence of actions (i.e., *true concurrency*). The theory of Petri Nets, which was the first well-established theory of concurrency, is an elegant generalization of classic automata theory such that the concept of concurrently occurring transitions can be expressed.

**A general model vs. special models.** In spite of promising progress in this direction (e.g., [GP93, MR96, Mil02]) an all-embracing model of concurrency has yet to emerge. According to Petri [Pet62] such a general model may attain a range of application comparable to that of physics. As argued in [Mil90], however, even after the discovery of it, we shall need to choose different special models for different applications. Here is an analogy from [Lam90]: Newtonian mechanics is not a suitable framework for describing the flow of fluids, for which one needs a theory containing mathematical concepts corresponding to friction and viscosity. Concurrency, as physics, is a field with a myriad of aspects for which we may require different terms of discussion and analysis.

### 1.1.4 Model Extensions

Science has made progress by extending well established theories to capture new and wider phenomena. For instance, computability theory was initially concerned only with functions on the natural numbers but it was later extended to deal with functions on the reals [Grz57]. Also, classical logic was extended to various modal logics to study reasoning involving modalities such as possibility, necessity and temporal progression. Another example of relevance to this dissertation is automata theory, initially confined to finite sequences, but later generalized to reason about infinite ones as in Büchi automata theory [Buc62].

Similarly, several mature models of concurrency have been extended to treat additional issues. These extensions should not come as a surprise since the field is indeed large and subject to the advents of new technology.

One example of these additional issues is the notions of mobility and security which now pervade the informational world; none of the representative models mentioned above dealt with these notions. It was later found that a CCS extension, the π-calculus [Mil99], could treat mobility in a very satisfactory way. A further extension, the spi-calculus [AG97], was also designed to model security.

Another prominent example is the notion of *time*, central to this dissertation. This notion not only is a fundamental concept in concurrency but also in science at large. Just like modal extensions of logic for temporal progression study time in logic reasoning, theories of concurrency were extended to study
time in concurrent activity. For instance, neither CCS, CSP nor Petri Nets, in
their basic form, were concerned with temporal behavior but they all have been
extended to incorporate an explicit notion of time. Namely, Timed CCS [Yi91],
Timed CSP [RR88], Timed ACP [BB91] and Timed Petri Nets [Zub80].

The above evidence suggests that for developing a theory of some phenom-
ena, in the present case discrete-timed systems, a research strategy is to see
if it can arise as an extension (or generalization) of an existing mature model.
This way we can benefit from the development and mathematical apparatus of
a well-established model. This strategy, at least with a little hindsight, is what
led to the temporal model presented in this dissertation; the ntcc calculus.

1.1.5 This Dissertation: A Temporal Extension of CCP

Saraswat’s concurrent constraint programming (ccp) model [SRP91] is a well-
established formalism for concurrency based upon the shared-variables com-
munication model. Its basic intuitions arise mostly from logic. In ccp, agents
can interact by adding (or telling) partial information in a medium, a so-called
store. Partial information is represented by constraints (e.g., \( x > 42 \)) on the
shared variables of the system The other way in which agents can interact is
by asking partial information to the store. This provides the synchronization
mechanism of the model; asking agents are suspended until there is enough
information in the store to answer their query.

As other mature models of concurrency, ccp has been extended to capture
aspects such as mobility [DRV98, Ré98, GP00], stochastic behavior [GJP99],
and most prominently time [SJG94a, SJG95, dBGM00, GJS98]. One such ex-
tension is the tcc model [SJG94a] which provides for discrete-timed and reactive
computations by allowing processes that can be constrained by unit delays and
time-out conditions. The tcc model is inherently deterministic and synchro-
nous.

This dissertation studies temporal concurrent constraint programming as a
model for timed systems by using a generalization of tcc. This generaliza-
tion is called the ntcc calculus and was originated in [NPV02b] by Mogens
Nielsen, Catuscia Palamidessi and the author. The ntcc calculus extends tcc
to model two notions not treated by tcc; nondeterminism and asynchrony.
Because of their ubiquity in concurrent systems, these notions are of the essence
for modeling process behavior faithfully. In fact, they allow ntcc to model
temporal behavior not expressible in tcc. Nondeterminism can provide, for
example, bounded-eventuality behavior (e.g “the gate must be open within 42
time units”). Asynchrony provides among others unbounded but finite-delays
behaviors (e.g., “emails are eventually delivered but without (knowing) exact
bounds on the delivery time”).

Development of ntcc. The development of the ntcc calculus presented
in this dissertation will justify its compliance with the criteria for models of
concurrency previously mentioned. Describing each item of these criteria wrt
ntcc gives a short description of the nature of the calculus.
• nttc is simple: It captures fundamental aspects of concurrency (i.e., discrete-time reactive computations, non-determinism, synchrony and asynchrony) whilst keeping a pleasant degree of simplicity.

• nttc is expressive: The expressiveness of nttc is illustrated by modeling several interesting applications (e.g., timed systems involving mutable structures, co-operative robotic devices and music applications).

• nttc is formal: It is founded on formal theories such as ccp, process calculi and first-order logic.

• nttc provides reasoning techniques: The techniques for reasoning about behavior of processes are the following:

  - A denotational semantics which interprets a given process as the set of sequences of actions it can potentially exhibit while interacting with arbitrary environments. This denotational semantics resembles those of CSP.

  - A process logic with an associated inference system than can be used much like the Hoare's program logic for sequential computation. The logic can be used to express required timed behaviors of processes, i.e., temporal specifications. The inference system can be used to prove that a process fulfills the specification.

  - Several equivalences, which are characterized operationally, to compare process behavior much like the behavioral equivalences for existing process calculi (e.g., bisimilarity and trace-equivalence).

  - A finite state automata representation of processes for a substantial fragment of the calculus. This makes available automatic tools for these automata for reasoning about nttc processes.

Timed ccp languages. In Chapter 8, this dissertation uses nttc to state new results of relevance to the ccp community as well as to extend, strengthen, and understand previous work in ccp (e.g., by proving and disproving some conjectures suggested in the literature). Some of these results are:

1. A classification according to the expressive power of several tcc languages found in the literature.

2. The decidability and undecidability of various natural behavioral equivalences for these languages.

3. A qualitative distinction between static and dynamic scoping which by itself helps to clarify the situation of the ccp family of languages.

The author believes that these results contribute to the understanding of the theory of timed ccp languages. Moreover, the methods used for the decidability results involve a systematic application of the classical Büchi automata theory [Buc62]. One may hope that such an application of standard automata
theoretic methods to timed ccp languages will provide a framework to perform further systematic investigations of these languages, e.g., algorithms and decision procedures for verifying that systems satisfy temporal properties.

**About the thesis.** The novelty of the ntcc calculus (and also of tcc) is that it combines in one framework an operational and algebraic view of processes based upon process calculi with a declarative view of processes based upon the modal logic of linear time (or linear-temporal logic) of [MP91]. In other words, the processes terms can be viewed at the same time as computing agents, algebraic terms and linear-temporal formulae. For example, the constructor $|$ can be thought of as parallel execution, a commutative operator and temporal-logic conjunction. At this point it is convenient to quote [Mil89]:

> I make no claim that everything can be done by algebra ... It is perhaps equally true that not everything can be done by logic; thus one of the outstanding challenges in concurrency is to find the right marriage between logic and behavioral approaches
> — Robin Milner.

In fact, the combination in one framework of the alternative views of processes mentioned above allows ntcc to benefit from the large body of techniques of both process calculi and linear-temporal logic (respectively, well established theories of concurrency and reactive discrete-timed computations). This will render the ntcc calculus suitable for the analysis and description of discrete-timed and reactive systems; the thesis of this dissertation.

**1.2 Background**

The ntcc calculus combines ideas of ccp, process calculi and temporal logic. This section gives a brief description of ccp and the basic concepts and issues from the development of process calculi and temporal logics that have influenced this work.

**1.2.1 Concurrent Constraint Programming**

In his seminal PhD thesis [Sar93], Saraswat proposed concurrent constraint programming as a model of concurrency based on the shared-variables communication model and a few primitive ideas taking root in logic. As informally
described in the next section, the ccp model elegantly combines logic concepts and concurrency mechanisms.

Concurrent constraint programming traces its origins back to Montanari’s pioneering work [Mon74] leading to constraint programming and Shapiro’s concurrent logic programming [Sha90]. The ccp model has received a significant theoretical and implementational attention: Saraswat, Rinard and Panangaden [SRP91] as well as De Boer, Di Pierro and Palamidessi [dBPP95a] gave fixed-point denotational semantics to ccp whilst Montanari and Rossi [RM94] gave it a (true-concurrent) Petri-Net semantics; De Boer, Gabrielli et al [dBGMP97] developed an inference system for proving properties of ccp processes; Smolka’s Oz [Smo95] as well as Haridi and Janson’s AKL [HJ90] programming languages are built upon ccp ideas.

Description of the model

A fundamental issue of the ccp model is the specification of concurrent systems in terms of constraints. A constraint is a first-order formula representing partial information about the shared variables of the system. For example, the constraint \( x + y > 42 \) specifies possible values for \( x \) and \( y \) (those satisfying the inequality). The ccp model is parameterized in a constraint system which specifies the constraints of relevance for the kind of system under consideration and an entailment relation \( \models \) between constraints (e.g., \( x + y > 42 \models x + y > 0 \)).

During computation, the state of the system is specified by an entity called the store where items of information about the variables of the system reside. The store is represented as a constraint and thus it may provide only partial information about the variables. This differs fundamentally from the traditional view of a store based on the Von Neumann memory model, in which each variable is assigned a uniquely determined value (e.g., \( x = 42 \) and \( y = 7 \)) rather than a set of possible values.

Some readers may feel uneasy as the notion of store in ccp suggests a model of concurrency with central memory. This is, however, an abstraction that simplifies the presentation of the model. The store can be distributed in several sites according to the agents that share the same variables (see [Sar93] for further discussions about this matter). Conceptually, the store in ccp is the medium through which agents interact with each other.

The ccp processes can update the state of the system only by adding (or telling) information to the store. This is represented as the (logical) conjunction of the constraint being added and the store representing the previous state. Hence, the update is not about changing the values of the variables but rather about ruling out some of the previously possible values. In other words, the store is monotonically refined.

Furthermore, processes can synchronize by asking information to the store. Asking is blocked until there is enough information in the store to entail (i.e., answer positively) their query. The ask operation is seen as determining whether the constraint representing the store entails the query.

A ccp computation terminates whenever it reaches a point, called resting or quiescent point, in which no more new information is added to the store. The
1.2. Background

final store, also called quiescent store (i.e., the store at the quiescent point), is the output of the computation.

Example 1.2.1. To make the description of the ccp model clearer, consider the simple ccp scenario illustrated in Figure 1.1. We have four agents (or processes) wishing to interact through an initially empty medium. Let us name them, starting from the upper rightmost agent in a clockwise fashion, A₁, A₂, A₃ and A₄, respectively. Suppose that they are scheduled for execution in the same order they were named.

This way A₁ moves first and tells the others through the medium that the temperature value is greater than 42 degrees but without specifying the exact value. In other words A₁ gives the others partial information about the temperature. This causes the addition of the item "temperature>42" to the previously empty store.

Now A₂ asks whether the temperature is exactly 50 degrees, and if so it wishes to execute a process P. From the current information in the store, however, it cannot be determined what the exact value of the temperature is. The agent A₂ is then blocked and so is the agent A₃ since from the store it cannot be determined either whether the temperature is between 0 and 100 degrees.

The turn is now for A₄ which tells that the temperature is less than 70 degrees. The store becomes "temperature > 42 \land temperature < 70". Now process A₃ can execute Q as its query is entailed by the information in the store. The other ask agent A₂ is doomed to be blocked forever unless Q adds enough information to the store to entail its query.

![Figure 1.1: A simple ccp scenario](image)

The CCP Process Language

In the spirit of process calculi, the language of processes in the ccp model is given with a reduced number of primitive operators or combinators. Rather than giving the actual syntax of the language, we content ourselves with describing the basic intuition that each construct embodies. So, in ccp we have:
• The tell action, for expressing tell operations (e.g., agent $A_1$ above).

• The ask action (or prefix action), for expressing an ask operation that prefixes another process; its continuation. E.g. the agent $A_2$ above.

• Parallel composition, which combines processes concurrently. For example the scenario in Figure 1.1 can be specified as the parallel composition of $A_1$, $A_2$, $A_3$ and $A_4$.

• Hiding (or locality), for expressing local variables that delimit the interface through which a process can interact with others.

• Summation, which expresses a disjunctive combination of agents to allow alternate courses of action.

• Recursion, for defining infinite behavior.

It is worth pointing out that without summation, the ccp model is deterministic in the sense that the quiescent or final store is always the same, independently from the execution order (scheduling) of the parallel components [SRP91].

1.2.2 Reactive Concurrent Constraint Programming

The tcc model introduced in [SJG94a] is an extension of ccp aimed at programming and modeling timed, reactive systems. This model, which has attracted growing attention during the last five years or so, elegantly combines ccp with ideas from the paradigms of Synchronous Languages [BG92, Hal98].

As any other model of computation, the tcc model makes an ontological commitment about computation. It emphasizes the view of reactive computation as proceeding deterministically in discrete time units (or time intervals). More precisely, time is conceptually divided into discrete intervals. In each time interval, a deterministic ccp process receives a stimulus (i.e. a constraint) from the environment, it executes with this stimulus as the initial store, and when it reaches its resting point, it responds to the environment with the final store. Also, the resting point determines a residual process, which is then executed in the next time interval.

This view of reactive computation is particularly appropriate for programming reactive systems such as robotic devices, micro-controllers, databases and reservation systems. These systems typically operate in a cyclic fashion; in each cycle they receive and input from the environment, compute on this input, and then return the corresponding output to the environment.

The fundamental move in the tcc model is to extend the standard ccp with delay and time-out operations. These operations are fundamental for programming reactive systems. 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.
1.2. Background

**Pre-emption and multi-form time.** In spite of its simplicity, the tcc extension to ccp is far-reaching. Many interesting temporal constructs can be expressed in tcc. In particular:

- **do P watching c.** This interrupt process executes P continuously until the item of information (e.g., a signal) c is present (i.e., entailed by the information in the store); when c is present P is killed from the next time unit onwards. This corresponds to the familiar `kill` command in Unix or clicking on the stop bottom of your favorite web browser.

- **S_cA_d(P).** This pre-emption process executes P continuously until c is present; when c is present P is suspended from the next time unit onwards. The process P is reactivated when d is present. This corresponds to the familiar (ctrl -Z, fg) mechanism in Unix.

- **time P on c.** This denotes a process whose notion of time is the occurrence of the item of information c. That is, P evolves only in those time intervals where c holds.

In general, tcc allows processes to be “clocked” by other processes. This provides meaningful pre-emption constructs and the ability of defining multiple forms of time instead of only having a unique global clock.

1.2.3 Process Calculi

This section describes some concepts from process calculi which are relevant to this dissertation. We do not intent to give an in-depth review of these calculi (the interested reader is referred to [Mil90]), but rather to describe those issues which influenced the development of ntcc in this dissertation. In fact, the order in which these issues are presented reflects the structure of this dissertation.

There are many different process calculi in the literature mainly agreeing in their emphasis upon algebra. The main representatives are CCS [Mil89], CSP [Ho85] and the process algebra ACP [BK85, BW90]. The distinctions among these calculi 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. In what follows we discuss some of these issues briefly.

The Language of Processes

A common feature of the languages of process calculi is that they pay special attention to economy. That is, there are few operators or combinators, each one with a distinct and fundamental role. Process calculi usually provide the following combinators:

- **Action,** for representing the occurrence of atomic actions.

- **Product,** for expressing the parallel composition.
• Summation, for expressing alternate course of computation.

• Restriction (or Hiding), for delimiting the interaction of processes.

• Recursion, for expressing infinite behavior.

A process language. For the purposes of the exposition of the next sections we shall define a basic process language which exemplifies the above.

We presuppose an infinite set \( \mathcal{N} \) of names \( a, b, \ldots \) and then introduce a set of co-names \( \overline{\mathcal{N}} = \{ \overline{a} \mid a \in \mathcal{N} \} \) disjoint from \( \mathcal{N} \). The set of labels, ranged over by \( l \) and \( l' \), is \( \mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}} \). The set of actions \( \mathcal{Act} \), ranged over by the boldface symbols \( a \) and \( b \) extends \( \mathcal{L} \) with a new symbol \( \tau \). The action \( \tau \) is said to be the silent (internal or unobservable) action. The actions \( a \) and \( \overline{a} \) are thought of as being complementary, so we decree that \( \overline{a} = a \). The syntax of processes is given by:

\[
P, Q, \ldots ::= 0 \mid a.P \mid P + Q \mid P \parallel Q \mid P \setminus a \mid A(b_1, \ldots, b_n)
\]

Intuitive Description. The intuitive meaning of the process terms is as follows. The process 0 does nothing. \( a.P \) is the process which performs an atomic action \( a \) and then behaves as \( P \). The summation \( P + Q \) is a process which may behave as either \( P \) or \( Q \). \( P \parallel Q \) represents the parallel composition of \( P \) and \( Q \). Both \( P \) and \( Q \) can proceed independently but they can also synchronize if they perform complementary actions. The restriction \( P \setminus a \) behaves as \( P \) except that it cannot perform the actions \( a \) or \( \overline{a} \). The names \( a \) and \( \overline{a} \) are said to be bound in \( P \setminus a \). \( A(b_1, \ldots, b_n) \) denotes the invocation to a unique recursive definition of the form \( A(a_1, \ldots, a_n) \overset{\text{def}}{=} P_A \) where all the non-bound names of process \( P_A \) are in \( \{ a_1, \ldots, a_n \} \). Obviously \( P_A \) may contain invocations to \( A \). The process \( A(b_1, \ldots, b_n) \) behaves as \( P_A[b_1, \ldots, b_n/a_1, \ldots, a_n] \), i.e., \( P_A \) with each \( a_i \) replaced by \( b_i \) – with renaming of bounded names wherever necessary to avoid captures.

Semantics of Processes

The methods by which process terms are endowed with meaning may involve at least three approaches: operational, denotational and algebraic semantics. Traditionally, CCS and CSP emphasize the use of the operational and denotational method, respectively, whilst the emphasis of ACP is upon the algebraic method. This dissertation will use these approaches with special emphasis upon the operational approach.

Operational semantics. The methods was pioneered by Plotkin in his Structural Operational Semantics (SOS) work [Plo81]. An operational semantics interprets a given process term by using transitions (labeled or not) specifying its computational steps. A labeled transition \( P \xrightarrow{a} Q \) specifies that \( P \) performs \( a \) and then behaves as \( Q \). The relations \( \xrightarrow{a} \) are defined to be the smallest which obey the rules in Table 1.1. In these rules the transition below the line is to be inferred from those above the line.
1.2. Background

Table 1.1: An operational semantics for a process calculus.

The rules in Table 1.1 are easily seen to realize the intuitive description of processes given in the previous section. Let us describe some. The rules SUM₁ and SUM₂ say that the first action of \( P + Q \) determines which alternative is selected, the other is discarded. The rules for composition COM₁ and COM₂ describe the concurrent performance of \( P \) and \( Q \). The rule COM₃ describes a synchronizing communication between \( P \) and \( Q \). For recursion, the rule REC says that the actions of (an invocation) \( A\langle b_1, \ldots, b_n \rangle \) are just those that can be inferred by replacing every \( a_i \) with \( b_i \) in (the definition’s body) \( P_A \) where \( A(a_1, \ldots, a_n) \overset{\text{def}}{=} P_A \).

Behavioral equivalences. Having defined the operational semantics, we can now introduce some typical notions of process equivalence. Here we shall recall trace, failures and bisimilarity equivalences. Although these equivalences can be defined for both CSP and CCS, traditionally the first two are associated with CSP and the last one is associated with CCS.

We need some little notation: The empty sequence is denoted by \( \epsilon \). Given a sequence of actions \( s = a_1.a_2.\ldots \in \text{Act}^* \), define \( \overset{s}{\Rightarrow} \) as

\[
(\overset{\tau}{\rightarrow})^* \overset{a_1}{\rightarrow} (\overset{\tau}{\rightarrow})^* \ldots (\overset{\tau}{\rightarrow})^* \overset{a_n}{\rightarrow} (\overset{\tau}{\rightarrow})^*
\]

Notice that \( \overset{\epsilon}{\Rightarrow} = (\overset{\tau}{\rightarrow})^* \). We use \( P \overset{s}{\Rightarrow} \) to mean that there exists a \( P' \) s.t., \( P \overset{s}{\Rightarrow} P' \) and similarly for \( P \overset{s}{\rightarrow} \).
• **Trace equivalence.** This equivalence is perhaps the simplest of all. Intuitively, two processes are deemed trace equivalent if and only if they can perform exactly the same sequences of non-silent (or observable) actions. Formally, we say that $P$ and $Q$ are trace equivalent, written $P =_T Q$, if for every $s \in \mathcal{L}^*$,

$$P \xrightarrow{s} \iff Q \xrightarrow{s}.$$ 

A drawback of $=_T$ is that it is not sensitive to deadlocks. For example, let $P_1 = a.b.0 + a.0$ and $Q_1 = a.b.0$. Notice that $P_1 =_T Q_1$ but unlike $Q_1$, after doing $a$, $P_1$ can reach a state in which it cannot perform any action, i.e., a deadlock.

• **Failures equivalence.** This equivalence is more discriminating (stronger or finer) than trace equivalence. In particular it is sensitive to deadlocks.

A failure is a pair $(s, L)$ where $s \in \mathcal{L}^*$ (called a trace) and $L$ is a set of labels. Intuitively, $(s, L)$ is a failure of $P$ if $P$ can perform a sequence of observable actions $s$ evolving into a $P'$ in which no action from $L \cup \{\tau\}$ can performed.

Formally, we say that $(s, L)$ is a failure of $P$ if there exists $P'$ such that

1. $P \xrightarrow{s} P'$,
2. $P \xrightarrow{\tau}$ and
3. for all $l \in L$, $P \xrightarrow{l}$.

We then say that $P$ and $Q$ are failures-equivalent, written $P =_F Q$, iff they possess the same failures.

Notice that $=_F \subseteq =_T$ as a trace is part of a failure. To see the strict inclusion, notice that for the trace equivalent processes $P_1$ and $Q_1$ given in the previous point, we have $P_1 \neq_F Q_1$ as $P_1$ has the failure $(a, \{b\})$ but $Q_1$ does not. Another interesting example is the processes $P_2 = a.(b.0 + c.0)$ and $Q_2 = a.b.0 + a.c.0$. They have the same traces, however $P_2 \neq_F Q_2$ since $P_2$ has the failure $(a, \{c\})$ but $Q_2$ does not.

• **Bisimilarity.** Here we first recall the strong version of the equivalence. Intuitively, $P$ and $Q$ are strongly bisimilar if whenever $P$ performs an action $a$ evolving into $P'$ then $Q$ can also perform $a$ and evolve into a $Q'$ strongly bisimilar to $P'$, and similarly with $P$ and $Q$ interchanged.

The above intuition can be formalized as follows. A symmetric relation $B$ between process terms is said to be a strong bisimulation iff for all $(P, Q) \in B$,

$$\text{if } P \xrightarrow{a} P' \text{ then for some } Q', Q \xrightarrow{a} Q' \text{ and } (P', Q') \in B.$$ 

We say that $P$ is strongly bisimilar to $Q$, written $P =_{SB} Q$ iff there exists a strong bisimulation containing the pair $(P, Q)$.

A weaker version of strong bisimilarity, called weak bisimilarity or simply bisimilarity, abstracts away from silent actions. Bisimilarity can be obtained by replacing the transitions $\xrightarrow{a}$ above with the (sequences of
observable) transitions $\Rightarrow s$ where $s \in \mathcal{L}^*$. We shall use $=_B$ to stand for (weak) bisimilarity. Notice that $P =_B \tau P$ but $P \not\equiv_S \tau P$.

Bisimilarity is more discriminating than trace equivalence. It is easy to see that $=_B \subseteq =_T$. The usual example to see the strict inclusion is $P_2$ and $Q_2$ as given above. Also, bisimilarity is more discriminating than failures equivalence wrt the branching behavior (i.e., nondeterminism); take $P_3 = a.(b.c.0 + a.d.0)$ and $P_3 = a.b.c.0 + a.b.d.0$; they have the same failures but one can verify that $P_3 \neq_B Q_3$. However, failures equivalence is more discriminating than bisimilarity wrt divergence (i.e., the execution of infinite sequences of silent actions). Notice that the divergent process $\text{Div}$, with $\text{Div} \overset{\text{def}}{=} \tau.\text{Div}$, is bisimilar to the non-divergent $\tau.0$, however $\text{Div} \not\equiv_F \tau.0$ since $\tau.0$ has the failure $(\epsilon, \emptyset)$ but $\text{Div}$ does not.

**Denotational Semantics.** The method was pioneered by Strachey and provided with a mathematical foundation by Scott. A denotational semantics interprets processes by using a function $[.]$ which maps them into a more abstract mathematical object (typically, a structured set or a category). The map $[.]$ is *compositional* in that the meaning of processes is determined from the meaning of its sub-processes.

A strategy for defining denotational semantics advocated in works such as [Ho90] involves the identification of what can be observed of a process; what behavior is deemed relevant (e.g., failures, traces, divergence, deadlocks). A process is then equated with the set of observations that can be made of it. For example, if the observation is the traces of processes, the denotation of the prefix construct $a.P$ can be defined as

$$[a.P] = \{\epsilon\} \cup \{a.s \in \mathcal{L}^* \mid s \in [P]\}$$

and the denotation of the summation can be defined as

$$[P + Q] = [P] \cup [Q].$$

It easy to see that these denotations realize the operational intuition of traces; any trace of $a.P$ is either empty or it starts with $a$ followed by a trace of $P$; any trace of $P + Q$ is either a trace of $P$ or one of $Q$. Note that the compositional nature is illustrated by stating the denotations of $a.P$ and $P + Q$ in terms of those of $P$ and $Q$.

Once the denotation has been defined one may ask whether it is in complete agreement with a corresponding operational notion. For example, for the trace denotation one would like the following correspondence wrt the operational notion of trace equivalence,

$$[P] = [Q] \text{ iff for all contexts } C, C[P] =_T C[Q]$$

(A *context* is an expression with a hole $[.]$ such that placing a process in the hole produces a well-formed process term, e.g., if $C = R \parallel [.]$ then $C[P] = R \parallel P$.) If a denotational-operational agreement like the one above can be proven, we say that the denotation is *fully abstract* [Mil73] wrt the chosen operational notion.
Denotational semantics are more abstract than the operational ones in that they generally distort themselves from any specific implementation. However, the operational semantics approach is, in some informal sense, more elemental in that when developing a denotational semantics one usually has an operational semantics in mind.

**Algebraic semantics.** This method has been advocated by Baeten and Weijland [BW90] as well as Bergstra and Klop [BK85]. An algebraic semantics attempts to give meaning by stating a set of laws (or axioms) equating process terms. The processes and their operations are then interpreted as structures that obey these laws. As remarked by Baeten and Weijland [BW90], the algebraic approach answers the question “What is a process?” with a seemingly circular answer: “A process is something that obeys a certain set of axioms...for processes”.

As an example consider the following axioms for parallel composition:

\[ P \parallel 0 \equiv P, \quad P \parallel Q \equiv Q \parallel P, \quad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \]

In other words parallel composition is seen as a commutative, associative operator with 0 being the unit. Notice that the above axioms basically equate processes that are the same except for irrelevant syntactic differences, thus one may expect that any reasonable notion of equivalence validates them. But consider the following distribution axiom

\[ a.(P + Q) \equiv a.P + a.Q \]

This axiom is valid if we are content with trace equivalence, but not in general (e.g., it does not hold for failures equivalence or bisimilarity).

Given a set of algebraic laws, one may be interested in looking into the correspondence with a denotational semantics or with some operational notion of equivalence. An interesting property is whether the equalities derived from the laws are exactly those which hold for a natural notion of process equivalence. If this property holds, the set of algebraic laws is said to be complete wrt the notion of process equivalence under consideration.

In the algebraic approach one can simply *postulate* process equalities while in the operational (or denotational) approach one would need to *prove* them. On the advantages of postulation Russell [Rus19] remarked the following:

*The method of postulation has many advantages: they are the same as the advantages of theft over honest toil*

— Bertrand Russell

Algebraic semantics, however, is a convenient framework for the study of process equivalences; postulating a set of laws, and then investigating the consistency of that set and what process equivalence it produces. Some frameworks (e.g., [Mil99]) combine the operational semantics with the algebraic one by, for example, considering processes modulo the equivalence produced by a set of axioms.
1.2. Background

Specification and Process Logics

One often is interested in verifying whether a given process satisfies a property, i.e., a specification. But process terms themselves specify behavior, so they can also be used to express specifications. Then this verification problem can be reduced to establishing whether the process and the specification process are related under some behavioral equivalence (or pre-order).

Hennesy-Milner’s modal logic. Another way of expressing process specifications is by using a process logic. One such a logic is the Hennesy-Milner’s modal logic. The basic syntax of formulae is given by:

\[ F := \text{true} \mid \text{false} \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \langle K \rangle F \mid [K] F \]

where \( K \) is a set of actions. Intuitively, the modality \( \langle K \rangle F \), called possibility, asserts (of a given \( P \)) that: It is possible for \( P \) to do \( a \in K \) and then evolve into a \( Q \) which satisfies \( F \). The modality \( [K]P \), called necessity, expresses that if \( P \) can do \( a \in K \) then it must thereby evolve into a \( Q \) which satisfies \( F \).

Formally, the compliance of \( P \) with the specification \( F \), written \( P \models F \), is recursively given by:

\[
\begin{align*}
P &\not\models \text{false} \\
P &\models \text{true} \\
P &\models F_1 \land F_2 \quad \text{iff} \quad P \models F_1 \text{ and } P \models F_2 \\
P &\models F_1 \lor F_2 \quad \text{iff} \quad P \models F_1 \text{ or } P \models F_2 \\
P &\models \langle K \rangle F \quad \text{iff} \quad \text{for some } Q, P \xrightarrow{a} Q, a \in K \text{ and } Q \models F \\
P &\models [K] F \quad \text{iff} \quad \text{if } P \xrightarrow{a} Q \text{ and } a \in K \text{ then } Q \models F
\end{align*}
\]

As an example consider our familiar trace equivalent (but not bisimilar equivalent) processes \( P_1 = a.(b.0 + c.0) \) and \( P_2 = a.b.0 + a.c.0 \). Notice that the formulae

\[ F = \langle \{a\} \rangle (\langle \{b\} \rangle \text{true} \land \langle \{c\} \rangle \text{true}) \]

discriminate among them, i.e. \( P_1 \models F \) but \( P_2 \not\models F \). In fact the discriminating power of this logic wrt a finite processes (i.e., recursion-free processes) coincides with strong bisimilarity (see [Sti98]). That is, two finite processes are strongly bisimilar iff they satisfy the same formulae in the Hennesy-Milner’s logic.

Temporal logics. The above logic can express local properties such as “an action must happen next” but it cannot express long-term properties such as “an action eventually happens”. This kind of property, which falls into the category of liveness properties (expressing that “something good eventually happens”), and also safety properties (expressing that “something bad never happens”) have been found to be useful for reasoning about concurrent systems. The modal logics attempting to capture properties of the kind above are often referred to as temporal-logics.

Temporal logics were introduced into computer science by Pnueli [Pnu77] and thereafter proven to be a good basis for specification as well as for (automatic and machine-assisted) reasoning about concurrent systems. Temporal
logics can be classified into linear and branching time logics. In the linear case at each moment there is only one possible future whilst in the branching case at each moment time may split into alternative futures.

Below we consider a very simple example of a linear-time temporal logic based on [MP91]. The syntax of the formulae is given by

\[ F := \text{true} \mid \text{false} \mid L \mid F_1 \lor F_2 \mid F_1 \land F_2 \mid \Diamond F \mid \Box F \]

where \( L \) is a set of non-silent actions. The formulae of this logic express properties of sequences of non-silent actions; i.e. traces. For the sake of uniformity, we are interested only in infinite traces. Intuitively, the modality \( \Diamond F \), pronounced \( \text{eventually } F \), asserts of a given trace \( s \) that at some point in \( s, F \) holds. Similarly, \( \Box F \), pronounced \( \text{always } F \), asserts of a given trace \( s \) that in every point of \( s, F \) holds.

The models of the formulae are taken to be infinite sequence of actions; elements of \( \text{Act}^\omega \). Formally, the infinite sequence of actions \( s = a_1.a_2 \ldots \) satisfies (or is a model of) \( F \), written \( s \models F \), iff \( \langle s, 1 \rangle \models F \), where

\[
\begin{align*}
\langle s, i \rangle &\models \text{true} \\
\langle s, i \rangle &\not\models \text{false} \\
\langle s, i \rangle &\models L \quad \text{iff} \quad a_i \in L \cup \tau \\
\langle s, i \rangle &\models F_1 \lor F_2 \quad \text{iff} \quad \langle s, i \rangle \models F_1 \text{ or } \langle s, i \rangle \models F_2 \\
\langle s, i \rangle &\models F_1 \land F_2 \quad \text{iff} \quad \langle s, i \rangle \models F_1 \text{ and } \langle s, i \rangle \models F_2 \\
\langle s, i \rangle &\models \Box F \quad \text{iff} \quad \text{for all } j \geq i \langle s, j \rangle \models F \\
\langle s, i \rangle &\models \Diamond F \quad \text{iff} \quad \text{there is a } j \geq i \text{ s.t. } \langle s, j \rangle \models F
\end{align*}
\]

Intuitively, \( P \) satisfies a linear-temporal specification \( F \), written \( P \models F \), iff all of its traces are models of \( F \). Recall, however, that the traces are finite sequences of non-silent actions. But since formulae say nothing about silent actions, we can just interpret a finite trace \( s \) as the infinite sequence \( \hat{s} = s.(\tau^\omega) \) which results from \( s \) followed by infinitely many silent actions. This leads to the definition: \( P \models F \) iff whenever \( P \xrightarrow{\hat{s}} \) then \( \hat{s} \models F \).

Let us consider the definitions \( A(a, b, c) \overset{\text{def}}{=} a.(b.A \langle a, b, c \rangle + c.A \langle a, b, c \rangle) \) and \( B(a, b, c) \overset{\text{def}}{=} a.b.B \langle a, b, c \rangle + a.c.B \langle a, b, c \rangle \). Notice that the trace equivalent processes \( A \langle a, b, c \rangle \) and \( B \langle a, b, c \rangle \) satisfy the formula \( \Box \Diamond (b \lor c) \); i.e. they always eventually do \( b \) or \( c \). In general, for every two processes (finite or infinite) if they are trace equivalent then they satisfy exactly the same formulae of this temporal logic. The other direction does not hold in general since the logic is not powerful enough to express, for example, facts about the immediate (or next) future. Take the processes \( a.a.0 \) and \( a.0 \); they are not trace equivalent, but they satisfy the same formulae in this simple logic.

**Analyzing Equivalences: Decidability and Congruence Issues**

Much work in the theory of process calculi, and concurrency in general, involves the analysis of process equivalences. Let us say that our equivalence under consideration is denoted by \( \sim \). Two typical questions that arise are:

1. Is \( \sim \) decidable?
2. Is \( \sim \) a congruence?

The first question refers to the issue as to whether there can be an algorithm that fully determines (or decides) for every \( P \) and \( Q \) if \( P \sim Q \) or \( P \not\sim Q \). Since most process calculi can model Turing machines most natural equivalences are therefore undecidable. So, the interesting question is rather for what subclasses of processes is the equivalence decidable. For example, bisimilarity is undecidable for full CCS, but decidable for finite state processes (of course) and also for the families of infinite state processes including context-free processes [CHS92], pushdown processes [Sti96] and basic parallel processes [CHM93]. Obviously, the decidability of an equivalence leads to another related issue: the complexity of verifying the equivalence.

The second question refers to the issue as to whether the fact that \( P \) and \( Q \) are \((\sim)\) equivalent implies that they are still \((\sim)\) equivalent in any context. The equivalence \( \sim \) is a congruence if \( P \sim Q \) implies \( C[P] \sim C[Q] \) for every context \( C \) (as said before, a context \( C \) is an expression with a hole [ ] such that placing a \( P \) in the hole yields a process term). The congruence issue is fundamental for algebraic as well as practical reasons; one may not be content with having \( P \sim Q \) equivalent but \( R \parallel P \not\sim R \parallel Q \).

For example, trace equivalence and strong bisimilarity for the process language here considered is a congruence (see [Mil90]) but weak bisimilarity is not because is not preserved by summation contexts. Notice that we have \( b.0 =_B \tau.b.0 \), but \( a.0 + b.0 \neq_B a.0 + \tau.b.0 \). In this case new questions arise: In what restricted sense is the equivalence a congruence? What contexts is the equivalence preserved by? What is the closest congruence to the equivalence? The answer to these questions may lead to a re-formulation of the operators. For instance, the problem with weak bisimilarity can be avoided by using a somewhat less liberal summation called guarded-summation (see [Mil99]).

### Process Calculi Variants

Given a process calculus it makes sense to consider variants of it (e.g., subclasses of processes, new process constructs, etc) to seek for simpler presentations of the calculus or different applications of it. Having these variants one can ask, for example, whether the process equivalences become simpler or harder to analyze (as argued in the previous section) or whether there is loss or gain of expressive power.

To compare expressive power one has to agree on what it means for a variant to be as expressive as the other. A natural way of doing this is by comparing wrt some process equivalence: If for every process \( P \) in one variant there is a \( Q \) in the other equivalent to \( P \) then we say that the latter variant is as expressive (wrt to the equivalence under consideration) as the former one.

Several studies of variants of CCS and their relative expressive power have been reported in [AM02]. Also several variants of the \( \pi \)-calculus (itself a generalization of CCS) have been compared wrt weak-bisimilarity (see [SW01]). An interesting result is that the \( \pi \) calculus construction \( !P \) whose behavior is expressed by the law \( !P \equiv P \parallel !P \) can replace recursion without loss of expressive
power. This is rather surprising since the syntax of !P and its description are so simple. Other interesting result is that of Palamidessi [Pal97] showing that under some reasonable assumptions the asynchronous version of the \( \pi \)-calculus is strictly less expressive than the synchronous one.

1.3 Character and Organization

In this section we first argue for the main reasons that led to the study of temporal concurrent constraint programming by using the ntcc calculus rather that its direct predecessor the tcc model. We then describe the way this study is presented in this dissertation.

1.3.1 Nondeterminism and Asynchrony

Due to its ontological commitment about computation the tcc model is not meant for the specification of non-deterministic or asynchronous temporal behavior. As said before, patterns of temporal behavior such as “the system must output \( c \) within the next \( t \) time units” or “the message must be delivered but there is no bound in the delivery time” cannot be expressed within the model. It also rules out the possibility of choosing one among several alternatives as an output to the environment. The task of zigzagging (see Chapter 6), in which a robot can unpredictably choose its next move, is but one example where non-determinism is useful.

From a modeling perspective, given a concurrent system, non-deterministic constructs can be used for modeling its unpredictable responses to the environment as well as unpredictable interactions among its processes. Asynchrony allows to model autonomous processes that respond to the environment at undetermined relative speeds. The descriptions of such components may say nothing about their speed except that they eventually proceed, thus reflecting the well-known principle of fairness.

From a programming perspective, a benefit of allowing the specification of non-deterministic behavior is to free programmers from the necessity of coping with issues that are irrelevant to the problem specification. Dijkstra’s language of guarded commands, for example, uses a nondeterministic construction to help free the programmer from over-specifying a method of solution. As pointed out by Winskel [Win93], a disciplined use of nondeterminism can lead to a more straightforward presentation of programs. This view is consistent with the declarative flavor of ccp: The programmer specifies by means of constraints the possible values that the program variables can take, without being required to provide a computational procedure to enforce the corresponding assignments. Constraints state what is to be satisfied but not how.

Justified in the above, the ntcc calculus provides a guarded-choice operator for modeling non-deterministic behavior and unbounded finite-delay operator for asynchronous behavior. Computation in ntcc progresses as in tcc, except for the non-determinism induced by the new operators. The use of guarded-choice for modeling non-determinism can be found in CCS as presented in [Mil99] and also in the ccp model itself as present in [SJG94a]. The use of an unbounded
but finite-delay operator for modeling asynchrony is based on Milner's Calculus for Synchrony and Asynchrony SCCS [Mil92].

1.3.2 Organization

In what follows we describe the structure of this dissertation which, as said before, follows the structure of the background section on process calculi. Each chapter concludes with a summary of its contents and a discussion about related work. Frequently used notational conventions and terminology are summarized in the Index.

Chapters 2-3. The second chapter is an informal introduction to the ntcc calculus with examples. These examples also give a flavor of the range of application of the calculus. The third chapter introduces the calculus formally. It first gives a SOS operational semantics to ntcc along the lines of that for process calculi. Then it introduces the kind of process behavior deemed observable and the equivalences these observations induce. The first such an observable behavior, called input-output behavior, is the infinite sequences of input-output interactions in which a process can engage with an environment. The second, called default output behavior, focus on the output behavior of processes in the absence of an external environment. The third, called strongest-postcondition, focus on the output behavior in the presence of arbitrary environments.

Chapter 4. The declarative nature of ntcc comes to the surface in this chapter by considering the denotational characterization of the strongest postcondition, as defined by [dBGMP97] for ccp, and extended to a timed setting. It is shown that the elegant model based on closure operators, developed by [SRP91] for deterministic ccp, can be extended for ntcc without losing its essential simplicity. It is shown that such an extension presents new technical problems to deal with due to the presence of nondeterminism and the unless operator. It is also shown that full-abstraction holds for a class of processes (and process contexts) called locally independent. These are the processes in which the choice and unless constructs contain no bound variables in their guards. It turns out that the local-independence condition for the choice operator strictly subsumes the so-called restricted choice condition considered by [FGMP97]. Restricted choice means that in every choice the guards are either pairwise mutually exclusive or equal – e.g., blind (or internal) choice falls into this category.

Chapter 5. The logical nature of ntcc comes to the surface in this chapter by considering its relationship with linear-temporal logic. It is shown that all the operators of ntcc correspond to temporal logic constructs like the operators of ccp correspond to (classical) logic constructs. Furthermore, this chapter introduces a sound inference system for proving linear temporal properties of ntcc similar to that of [dBGMP97] for untimed ccp. The system is proven (relatively) complete wrt locally independent processes. An existing proof system for tcc in [SJG94a], whose underlying logic is intuitionistic rather than classical, is complete for hiding (and recursion) free tcc processes only.
Chapter 6. This chapter illustrates the expressive power of ntcc by describing several applications of the calculus. These applications involve the modeling and analysis of mutable data structures such as cells and examples involving timed systems (RCX™ controllers), multi-agent systems (the Predator/Prey game), and musical applications (generation of rhythms patterns and controlled improvisation).

Chapter 7. This chapter relates the various process equivalences introduced in Chapter 3 and addresses their decidability and congruence issues. The equivalences are proven to be decidable for a substantial fragment of the calculus. These decidability results involve systematic translations of ntcc processes into Büchi automata. The decidability for the complete calculus is left open. The closest congruences included in the equivalences are characterized by identifying interesting families of “distinguishing” contexts. For example, one characterization reduces the question of whether $P$ and $Q$ are output congruent to whether $C[P]$ is output-equivalent to $C[Q]$ where $C$ is a particular context for which there is an effective construction, given $P$ and $Q$ (thus reducing the decidability of the congruence to the decidability of the equivalence). If the underlying set of constraints is finite, such context is universal, meaning that it is the same for any $P$ and $Q$. The other characterizations state analogous results.

Chapter 8. Several tcc languages differing in their way of expressing infinite behavior have been proposed in the literature. This chapter studies the expressive power of a few fundamental representatives of these languages by considering them as variants of the ntcc calculus. In particular, the following is shown:

1. recursive procedures with parameters can be encoded into parameterless recursive procedures with dynamic scoping, and vice-versa,

2. replication can be encoded into parameterless recursive procedures with static scoping, and vice-versa, and

3. the languages from (1) are strictly more expressive than the languages from (2).

Furthermore, the chapter shows that behavioral equivalence is undecidable for the languages from (1), but decidable for the languages from (2). The undecidability result holds even if the process variables take values from a fixed finite domain while the decidability results holds for arbitrary domains. This chapter also introduces a bisimulation proof technique for the input-output process equivalence.

Chapter 9. This chapter discusses related work, gives some concluding remarks and proposes future work.
1.4 Contributions

Most of the material of this dissertation has been previously reported in the following works. The unpublished material will be explicitly mentioned in each chapter.


The main contributions of this paper are included in Chapter 7.


The main contributions of this paper are included in Chapters 4-6.

**Proceedings of International Conferences.**


The main contributions of this paper are included in Chapter 8.


The some of main contributions of this paper are given in Chapter 6.


This abstract describes temporal concurrent constraint programming and reports on some of the work presented in Chapters 5-7.


This paper is the first introduction to the ntcc calculus.
Chapter 2

Intuitive Description of ntcc

*One should not increase, beyond what is necessary, the number of entities required to explain anything*
— William of Occam

In this chapter we introduce the basic ideas underlying the ntcc calculus in an informal way. We shall begin by introducing the notion of a constraint system, which is central to concurrent constraint programming. We then describe the basic process constructs by means of examples. Finally, we shall describe some convenient derived constructs.

2.1 Intuitive Description of Constraint Systems

The ntcc processes are parametric in a *constraint system*. A constraint system provides a *signature* from which syntactically denotable objects called *constraints* can be constructed and an *entailment relation* $\models$ specifying interdependencies between these constraints.

A constraint represents a piece of information (or *partial information*) upon which processes may act. For instance, processes modeling temperature controllers may have to deal with partial information such as $42 < t_{sensor} < 100$ expressing that the sensor registers an unknown (or not precisely determined) temperature value between 42 and 100. The inter-dependency $c \models d$ expresses that the information specified by $d$ follows from the information specified by $c$, e.g., $(42 < t_{sensor} < 100) \models (0 < t_{sensor} < 120)$.

We can set up the notion of constraint system by using first-order logic. Let us suppose that $\Sigma$ is a signature (i.e., a set of constants, functions and predicate symbols) and that $\Delta$ is a consistent first-order theory over $\Sigma$ (i.e., a set of sentences over $\Sigma$ having at least one model). Constraints can be thought of as first-order formulae over $\Sigma$. We can then decree that $c \models d$ if the implication $c \Rightarrow d$ is valid in $\Delta$. This gives us a simple and general formalization of the notion of constraint system as a pair $(\Sigma, \Delta)$.

In the examples below we shall assume that, in the underlying constraint system, $\Sigma$ is the set $\{=, <, 0, 1 \ldots \}$ and $\Delta$ is the set of sentences over $\Sigma$ valid on the natural numbers.
2.2 Intuitive Description of Processes

We now proceed to describe with examples the basic ideas underlying the behavior of ntcc processes. For this purpose we shall model simple behavior of controllers such as Programmable Logic Controllers (PLC's) and RCX bricks.

PLC's are often used in timed systems of industrial applications [Die00], whilst RCX bricks are mainly used to construct autonomous robotic devices [LP99, HS00]. These controllers have external input and output ports. One can attach, for example, sensors of light, touch or temperature to the input ports, and motors, lights or alarms to the output ports. Typically PLC's and RCX bricks operate in a cyclic fashion. Each cycle consist of receiving an input from the environment, computing on this input, and returning the corresponding output to the environment.

Our processes will operate similarly. Time is conceptually divided into discrete intervals (or time units). In a particular time interval, a process $P_i$ receives a stimulus from the environment (see Equation 2.1 below). The stimulus is some piece of information, i.e., a constraint. The process $P_i$ executes with this stimulus as the initial store, and when it reaches its resting point (i.e., a point in which no further computation is possible), it responds to the environment with a resulting store $d_i$. Also the resting point determines a residual process $P_{i+1}$, which is then executed in the next time interval.

The following sequence illustrates the stimulus-response interactions between an environment that inputs $c_1, c_2, \ldots$ and a process that outputs $d_1, d_2, \ldots$ on such inputs as described above.

$$P_1 \xrightarrow{(c_1,d_1)} P_2 \xrightarrow{(c_2,d_2)} \ldots P_i \xrightarrow{(c_i,d_i)} P_{i+1} \xrightarrow{(c_{i+1},d_{i+1})} \ldots \quad (2.1)$$

2.2.1 Communication: Telling and Asking Information

The ntcc processes communicate with each other by posting and reading partial information about the variables of system they model. The basic actions for communication provide the telling and asking of information. A tell action adds a piece of information to the common store. An ask action queries the store to decide whether a given piece of information is present in it. The store as a constraint itself. In this way addition of information corresponds to logic conjunction and determining presence of information corresponds to logic implication.

The tell and ask processes have respectively the form

$$\text{tell}(c) \quad \text{and} \quad \text{when } c \text{ do } P \quad (2.2)$$

The only action of a tell process $\text{tell}(c)$ is to add, within a time unit, $c$ to the current store $d$. The store then becomes $d \land c$. The addition of $c$ is carried out even if the store becomes inconsistent, i.e., $(d \land c) = \text{false}$, in which case we can think of such an addition as generating a failure.

Example 2.2.1. Suppose that $d = (\text{motor1.speed} > \text{motor2.speed})$. Intuitively, $d$ tells us that the speed of motor one is greater than that of motor two.
2.2. Intuitive Description of Processes

It does not tell us what the specific speed values are. The execution in store $d$ of process

$$\text{tell(motor}_2\text{.speed > 10)}$$

causes the store to become $($motor$_1\text{.speed} > \text{motor}_2\text{.speed} > 10)$ in the current time interval, thus increasing the information we know about the system - we now know that both speed values are greater than 10.

Notice that in the underlying constraint system $d \models \text{motor}_1\text{.speed} > 0$, therefore the process

$$\text{tell(motor}_1\text{.speed = 0)}$$
in store $d$ causes a failure. \hfill \square

The process \texttt{when c do P} performs the action of asking $c$. If during the current time interval $c$ can eventually be inferred from the store $d$ (i.e., $d \models c$) then $P$ is executed within the same time interval. Otherwise, \texttt{when c do P} is precluded from execution in any future time interval (i.e., it becomes constantly inactive).

\textbf{Example 2.2.2}. Suppose that $d = (\text{motor}_1\text{.speed} > \text{motor}_2\text{.speed})$ is the store. The process

$$P = \texttt{when motor}_1\text{.speed} > 0 \texttt{ do Q}$$

will execute $Q$ in the current time interval since $d \models \text{motor}_1\text{.speed} > 0$, by contrast the process

$$P' = \texttt{when motor}_1\text{.speed} > 10 \texttt{ do Q}$$

will not execute $Q$ unless more information is added to the store, during the current time interval, to entail $\text{motor}_1\text{.speed} > 10$.

The intuition is that any process in $d = (\text{motor}_1\text{.speed} > \text{motor}_2\text{.speed})$ can execute a given action if and only if it can do so whenever variables $\text{motor}_1\text{.speed}$ and $\text{motor}_2\text{.speed}$ are set to arbitrary values satisfying $d$. For example, the process $P$ above executes $Q$ if $\text{motor}_1\text{.speed}$ and $\text{motor}_2\text{.speed}$ take on any value satisfying $d$. \hfill \square

The above example illustrates the partial information allows us to model the actions that a system can perform, regardless of the alternative values a variable may assume, as long they comply with the constraint representing the store.

\subsection*{2.2.2 Nondeterminism}

As argued above, partial information allows us to model behavior for alternative values that variables may take on. In concurrent systems it is often convenient to model behavior for \textit{alternative courses} of action, i.e., nondeterministic behavior.
We generalize the processes of the form \textbf{when} \( c \) \textbf{do} \( P \) described above to guarded-choice summation processes of the form

\[
\sum_{i \in I} \textbf{when} \ c_i \ \textbf{do} \ P_i
\]

(2.3)

where \( I \) is a finite set of indices. The expression \( \sum_{i \in I} \textbf{when} \ c_i \ \textbf{do} \ P_i \) represents a process that, in the current time interval, \textit{must non-deterministically} choose a process \( P_j \) \((j \in I)\) whose corresponding constraint \( c_j \) is entailed by the store. The chosen alternative, if any, precludes the others. If no choice is possible during the current time unit, all the alternatives are precluded from execution.

In the following example we shall use “+” for binary summations.

\textbf{Example 2.2.3.} Often RCX programs operate in a set of simple stimulus-response rules of the form \textbf{IF} \( E \) \textbf{THEN} \( C \). The expression \( E \) is a condition typically depending on the sensor variables, and \( C \) is a command, typically an assignment. In [Fre99] these programs respond to the environment by choosing a rule whose condition is met and executing its command.

If we wish to abstract from the particular implementation of the mechanism that chooses the rule, we can model the execution of these programs by using the summation process. For example, the program operating in the set

\[
\begin{align*}
\{ & (\text{IF} \ \text{sensor}_1 > 0 \ \text{THEN} \ \text{motor}_1.\text{speed} := 2), \\
& (\text{IF} \ \text{sensor}_2 > 99 \ \text{THEN} \ \text{motor}_1.\text{speed} := 0) \}
\end{align*}
\]

corresponds to the summation process

\[
P = + \textbf{when} \ \text{sensor}_1 > 0 \ \textbf{do} \ \text{tell}(\text{motor}_1.\text{speed} = 2)
\]

\[
+ \textbf{when} \ \text{sensor}_2 > 99 \ \textbf{do} \ \text{tell}(\text{motor}_1.\text{speed} = 0).
\]

In the store \( d = (\text{sensor}_1 > 10) \), the process \( P \) causes the store to become \( d \land (\text{motor}_1.\text{speed} = 2) \) since \( \text{tell}^{\text{motor}_1.\text{speed} = 2} \) is chosen for execution and the other alternative is precluded. In the store \textbf{true}, \( P \) cannot add any information. In the store \( e = (\text{sensor}_1 = 10 \land \text{sensor}_2 = 100) \), \( P \) causes the store to become either \( e \land (\text{motor}_1.\text{speed} = 2) \) or \( e \land (\text{motor}_1.\text{speed} = 0) \). \( \square \)

\textbf{2.2.3 Parallel Composition}

We need a construct to represent processes acting \textit{concurrently}. Given \( P \) and \( Q \) we denote their parallel composition by the process

\[
P \parallel Q
\]

(2.4)

In one time unit (or interval) processes \( P \) and \( Q \) operate concurrently, “communicating” via the common store by telling and asking information.

\textbf{Example 2.2.4.} Let \( P \) be defined as in Example 2.2.3 and

\[
Q = + \textbf{when} \ \text{motor}_2.\text{speed} = 0 \ \textbf{do} \ \text{tell}(\text{motor}_2.\text{speed} = 0)
\]

\[
+ \textbf{when} \ \text{motor}_2.\text{speed} = 0 \ \textbf{do} \ \text{tell}(\text{motor}_1.\text{speed} = 0).
\]
Intuitively $Q$ turns off one motor if the other is detected to be off. The parallel composition $P \parallel Q$ in the store $d = (\text{sensor}_2 > 100)$ will, in one time unit, cause the store to become $d \land (\text{motor}_1.\text{speed} = \text{motor}_2.\text{speed} = 0)$. □

### 2.2.4 Local Behavior

Most process calculi have a construct to restrict the interface through which processes can interact with each other, thus providing for the modeling of local (or hidden) behavior. We introduce processes of the form

$$(\text{local } x) P$$ (2.5)

The process $(\text{local } x) P$ declares a variable $x$, private to $P$. This process behaves like $P$, except that all the information about $x$ produced by $P$ is hidden from external processes and the information about $x$ produced by other external processes is hidden from $P$.

**Example 2.2.5.** In modeling RCX or PLC's one uses "global" variables to represent ports (e.g., sensor and motors). One often, however, uses variables which do not represent ports, and thus we may find it convenient to declare such variables as local (or private).

Suppose that $R$ is a given process modeling some controller task. Furthermore, suppose that $R$ uses a variable $z$, which is set at random, with some unknown distribution, to a value $v \in \{0, 1\}$. Let us define the process

$$P = ( \sum_{v \in \{0, 1\}} \text{when } \text{true do } \text{tell}(z = v) ) \parallel R$$

to represent the behavior of $R$ under $z$'s random assignment.

We may want to declare $z$ in $P$ to be local since it does not represent an input or output port. Moreover, notice that if we need to run two copies of $P$, i.e., process $P \parallel P$, a failure may arise as each copy can assign a different value to $z$. Therefore, the behavior of $R$ under the random assignment to $z$ can be best represented as $P^r = (\text{local } z) P$. In fact, if we run two copies of $P^r$, no failure can arise from the random assignment to the $z$'s as they are private to each $P^r$.

The processes hitherto described generate activity within the current time interval only. We now turn to constructs that can generate activity in future time intervals.

### 2.2.5 Unit Delays and Time-Outs

As in the Synchronous Languages [BG92] we have constructs whose actions can delay the execution of processes. These constructs are needed to model time dependency between actions, e.g., actions depending on the absence or presence of preceding actions. Time dependency is an important aspect in the modeling of timed systems.
The unit-delay operators have the form

\[
\text{next } P \quad \text{and} \quad \text{unless } c \text{ next } P
\]  

(2.6)

The process \text{next } P represents the activation of \( P \) in the next time interval. The process \text{unless } c \text{ next } P is similar, but \( P \) will be activated only if \( c \) cannot be inferred from the resulting (or final) store \( d \) in the current time interval, i.e., \( d \not\models c \). The “unless” processes add time-outs to the calculus, i.e., they wait during the current time interval for a piece of information \( c \) to be present and if it is not, they trigger activity in the next time interval.

Notice that \text{unless } c \text{ next } P is not equivalent to \text{when } \lnot c \text{ do } \text{next } P \text{ since } d \not\models c \text{ does not necessarily imply } d \models \lnot c. \text{ Notice also that the process } Q = \text{unless } \text{false } \text{next } P \text{ is not the same as } R = \text{next } P \text{ since unlike } Q, \text{ even if the store contains false, } R \text{ will still activate } P \text{ in the next time interval (and the store in the next time interval may not contain false).}

Example 2.2.6. The process

\[
\text{when } \text{false } \text{do next tell(motor}_{1}\text{.speed} = \text{motor}_{2}\text{.speed} = 0)
\]

turns the motors off by decreeing that \text{motor}_{1}\text{.speed} = \text{motor}_{2}\text{.speed} = 0 in the next time interval if a failure takes place in the current time interval. Similarly, the process

\[
\text{unless } \text{false } \text{next } (\text{tell(motor}_{1}\text{.speed} > 0) \parallel \text{tell(motor}_{2}\text{.speed} > 0))
\]

makes the motors move at some speed in the next time unit, unless a failure takes place in the current time interval.

\[\square\]

2.2.6 Asynchrony

We now introduce a construct that, unlike the previous ones, can describe arbitrary (finite) delays. The importance of this construct is that it allows us to model asynchronous behavior across the time intervals.

We use the operator “∗” which corresponds to the unbounded but finite delay operator for synchronous CCS [Mil92]. The process

\[
\ast P
\]

(2.7)

represents an arbitrary long but finite delay for the activation of \( P \). Thus, \( \ast \text{tell}(c) \) can be viewed as a message \( c \) that is eventually delivered but there is no upper bound on the delivery time.

Example 2.2.7. The process \( S \) below specifies that motor \text{motor}_{1}, \text{ at some unpredictable point in time, is doomed to malfunction}

\[
S = \ast \text{tell(malfunction(motor}_{1}\text{.status}))}
\]

\[\square\]
2.2.7 Infinite Behavior

Finally, we need a construct to define infinite behavior. We shall use the operator "!" as a delayed version of the replication operator for the π-calculus [Mil99]. Given a process $P$, the process

$$!P$$

represents $P \parallel (\text{next } P) \parallel (\text{next next } P) \parallel \ldots$, i.e., unboundedly many copies of $P$, but one at a time. The process $!P$ executes $P$ in one time unit and persists in the next time unit.

**Example 2.2.8.** The process $R$ below repeatedly checks the state of $\text{motor}_1$. If a malfunction is reported, $R$ tells that $\text{motor}_1$ must be turned off.

$$R = !\text{ when malfunction(\text{motor}_1.status) do tell(\text{motor}_1.speed = 0)}$$

Thus, the process $$P = R \parallel S$$

for $S = *\text{ tell(malfunction(\text{motor}_1.status))}$ (Example 2.2.7) eventually tells that $\text{motor}_1$ is turned off.  

2.2.8 Some Derived Forms

We have informally introduced the basic process constructs of ntcp and illustrated how they can be used to model or specify system behavior. In this section we shall illustrate how they can be used to obtain some convenient derived constructs.

In the following we shall omit "\text{when true do}" if no confusion arises. The "blind-choice" process $\sum_{i \in I} \text{ when true do } P_i$, for example, can be written as $\sum_{i \in I} P_i$. We shall use $\prod_{i \in I} P_i$, where $I$ is finite, to denote the parallel composition of all the $P_i$'s. We use $\text{next}^n(P)$ as an abbreviation for $\text{next}((\ldots(\text{next}(P)\ldots))$, where $\text{next}$ is repeated $n$ times.

**Inactivity**

The process doing nothing whatsoever, $\text{skip}$ can be defined as an abbreviation of the empty summation $\sum_{i \in \emptyset} P_i$. This process corresponds to the inactive processes $0$ of CCS and $\text{STOP}$ of CSP. We should expect the behavior of $P \parallel \text{skip}$ to be the same as that of $P$ under any reasonable notion of behavioral equivalence.

**Abortion**

Another useful construct is the process $\text{abort}$ which is somehow to the opposite extreme of $\text{skip}$. Whilst having $\text{skip}$ in a system causes no change whatsoever, having $\text{abort}$ can make the whole system fail. Hence $\text{abort}$ corresponds to the $\text{CHAO}S$ operator in CSP. In Section 2.2.1 we mentioned that a tell process causes a failure, at the current time interval, if it leaves the store inconsistent.
Therefore, we can define `abort` as `tell(false)`, i.e., the process that once activated causes a constant failure. Therefore, any reasonable notion of behavioral equivalence should not distinguish between `P || abort` and `abort`.

### Asynchronous Parallel Composition

Notice that in `P || Q` both `P` and `Q` are forced to move in the current time unit, thus our parallel composition can be regarded as being a synchronous operator. There are situations where an asynchronous version of `"||"` is desirable. For example, modeling the interaction of several controllers operating concurrently where some of them could be faster or slower than the others at responding to their environment.

By using the star operator we can define a *(fair) asynchronous* parallel composition `P | Q` as

\[
(P || * Q) + (* P || Q)
\]

A move of `P | Q` is either one of `P` or one of `Q` (or both). Moreover, both `P` and `Q` are eventually executed (i.e., a fair execution of `P | Q`). This process corresponds to the asynchronous parallel operator described in [Mil92].

We should expect operator `"||"` to enjoy properties of parallel composition. Namely, we should expect `P | Q` to be the same as `Q | P` and `P | (Q | R)` to be the same as `(P | Q) | R`. Unlike in `P || skip`, however, in `P | skip` the execution of `P` may be arbitrary postponed, therefore we may want to distinguish between `P | skip` and `P`. Similarly, unlike in `P || abort`, in `P | abort` the execution of `abort` may be arbitrarily postponed. In a timed setting we may want to distinguish between a process that aborts right now and one that may do so sometime later after having done some work.

### Bounded Eventuality and Invariance

We may want to specify that a certain behavior is exhibited within a certain number of time units, i.e., *bounded eventuality*, or during a certain number of time units, i.e., *bounded invariance*. An example of bounded eventuality is “the light must be switched off within the next ten time units” and an example of bounded invariance is “the motor should not be turned on during the next sixty time units”.

The kind of behavior described above can be specified by using the bounded versions of `!P` and `*P`, which can be derived using summation and parallel composition in the obvious way. We define `!P` and `*P`, where `I` is a closed interval of the natural numbers, as an abbreviation for

\[
\prod_{i \in I} \text{next}^i P \quad \text{and} \quad \sum_{i \in I} \text{next}^i P
\]

respectively. Intuitively, `*_{[m,n]}P` means that `P` is eventually active between the next `m` and `m + n` time units, while `!_{[m,n]}P` means that `P` is always active between the next `m` and `m + n` time units.
Nondeterministic Time-Outs

The ntcc calculus generalizes processes of the form \text{when } c \text{ do } P \text{ by allowing nondeterministic choice over them. It would therefore be natural to do the same with processes of the form unless } c \text{ next } P. \text{ In other words, one may want to have a nondeterministic time-out operator}

$$\sum_{i \in I} \text{unless } c_i \text{ next } P_i$$

which chooses one \(P_i\) such that \(c_i\) cannot be eventually inferred from the store within the current time unit (if no choice is possible then the summation is precluded from future execution). Notice that this is not the same as having a blind-choice summation of the unless \(c_i\) next \(P_i\) operators. It is not difficult to see, however, that the behavior of such a nondeterministic time-out operator can be described by the ntcc process:

\[
(\text{local } I') \left( \prod_{i \in I} (\text{unless } c_i \text{ next } \text{tell}(i \in I')) \parallel \text{next } \sum_{i \in I} \text{when } i \in I' \text{ do } P_i \right)
\]

where \(i \in I'\) holds iff \(i\) is in the set \(I'\).

2.3 Summary and Related Work

In this chapter we informally described the basic ideas underlying ntcc. We described the notion of constraint systems to which ntcc processes are parametric. We also described with examples the basic and some derived ntcc process constructs. The basic constructs provide: the asking and telling of information, nondeterminism, parallel execution, local behavior, delays, time-outs, asynchrony and infinite behavior. The derived constructs provide: inactivity, abortion, asynchronous parallel execution, bounded asynchrony, bounded invariance and nondeterministic time-outs.

The ntcc calculus was introduced in [NPV02b] as an extension of the tcc model of [SJG94a]. The fundamental difference is that, unlike tcc, ntcc provides for the specification of non-determinism and asynchronous behavior. Another difference with the tcc model lies in the way of defining infinite behavior. The tcc model uses recursive definitions. Instead ntcc provides the replication operator thus providing a more algebraic view of process constructs. Another extension of the tcc model [SJG96] uses the "hence" operator which is closely related to replication. The process hence \(P\) behaves exactly as next ! \(P\). Chapter 8 addresses expressiveness issues of these alternative ways of defining infinite behavior.
Chapter 3

Operational Semantics

...I have come to an even firmer belief that operational semantics, since it can be set up with so few preconceptions, must be the touchstone for assessing mathematical models rather than the reverse

— Robin Milner

In the previous chapter we gave an intuitive description of ntc. In this chapter we shall make precise such a description. We shall begin by defining the notion of constraint system and the formal syntax of ntc. We shall then give meaning to the ntc processes by means of an operational semantics. The semantics, which resembles the reduction semantics of the π-calculus [Mil99], provides internal and external transitions describing process evolutions. The internal transitions describe evolutions within a time unit and thus they are regarded as being unobservable. In contrast, the external transitions are regarded as being observable as they describe evolution across the time units. We shall introduce some relevant notions of observable process behavior based on the external transitions. Such notions induce behavioral process equivalences that abstract from internal transitions and thus from internal behavior.

3.1 Constraint Systems

For our purposes it will suffice to consider the notion of constraint system based on first-order logic, as was done in [Smo94].

Definition 3.1.1 (Constraint Systems). A constraint system is a pair \((\Sigma, \Delta)\) where \(\Sigma\) is a signature specifying constants, functions and predicate symbols, and \(\Delta\) is a consistent first-order theory over \(\Sigma\) (i.e., a set of first-order sentences over \(\Sigma\) having at least one model).

Given a constraint system \((\Sigma, \Delta)\), let \(\mathcal{L}\) be the underlying first-order language \((\Sigma, \mathcal{V}, \mathcal{S})\). Here \(\mathcal{V}\) is a countable set of variables \(x, y, \ldots\), and \(\mathcal{S}\) is the set of logic symbols \(\neg, \land, \lor, \Rightarrow, \exists, \forall, \text{true}\) and \(\text{false}\) which denote logical negation, conjunction, disjunction, implication, existential and universal quantification, and the always true and always false predicates, respectively. Constraints, denoted by \(c, d, \ldots\), are first-order formulae over \(\mathcal{L}\). We say that \(c\)
entails $d$ in $\Delta$, written $c \models_\Delta d$ iff the formula $c \Rightarrow d$ is true in all models of $\Delta$. We write $\models$ instead of $\models_\Delta$ when $\Delta$ is unimportant or can be inferred from the context. For operational reasons, we shall require $\models$ to be decidable. We say that $c$ is equivalent to $d$, written $c \equiv d$, iff $c \models d$ and $d \models c$.

**Convention 3.1.2.** Henceforth, $C$ denotes the set of constraints modulo $\equiv$ under consideration 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 $x = y$.

We shall now introduce some examples of constraint systems. The classical example is the Herbrand (or finite tree) constraint system [Sar93].

**Definition 3.1.3 (Herbrand Constraint System).** The Herbrand constraint system is such that:

- $\Sigma$ is the set with infinitely many function symbols of each arity and equality $\equiv$.

- $\Delta$ is given by Clark’s Equality Theory with the schemas

  $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \Rightarrow x_1 = y_1 \land \ldots \land x_n = y_n$

  $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_n) \Rightarrow \text{false, if } f, g \text{ are distinct symbols}$

  $x = f(\ldots x \ldots) \Rightarrow \text{false}$.

The importance of the Herbrand constraint system is that it underlies conventional logic programming and most first-order theorem provers. Its value lies in the Herbrand Theorem, which reduces the problem of checking unsatisfiability of a first-order formula to the unsatisfiability of a quantifier-free formula interpreted over finite trees.

Another widely used constraint system is the finite-domain constraint system $\text{FD}$ defined in [HSD95]. In $\text{FD}$ variables are assumed to range over finite domains and, in addition to equality, we may have predicates that restrict the range of a variable to some finite set. The following is a simplified finite-domain constraint system.

**Definition 3.1.4 (A Finite-Domain Constraint System).** Let $n > 0$. Define $\text{FD}[n]$ as the constraint system such that:

- $\Sigma$ is given by the constants symbols $0, 1, \ldots, n - 1$ and the equality $\equiv$.

- $\Delta$ is given by the axioms of equational theory $x = x$, $x = y \Rightarrow y = x$, $x = y \land y = z \Rightarrow x = z$, and $v = w \Rightarrow \text{false}$ for each two different constants $v, w$ in $\Sigma$.

Intuitively $\text{FD}[n]$ provides a theory of variables ranging over a finite domain of values $\{0, \ldots, n - 1\}$ with syntactic equality over these values. Due to its simplicity we shall use $\text{FD}[n]$ in our impossibility results involving finite-domain assumptions.

The following is a somewhat more complex finite-domain constraint system.

**Definition 3.1.5 (Modular Arithmetic Constraint System).** Let $n > 0$. Define $\text{A}[n]$ as the constraint system such that:
• $\Sigma$ is given by \{0, 1, ..., $n - 1$, succ, pred, $+$, $\times$, $=$, $>$\}.

• $\Delta$ is the set of sentences valid in arithmetic modulo $n$.

The intended meaning of $A[n]$ is the natural numbers interpreted as in arithmetic modulo $n$. Due to the familiar operations it provides, we shall often assume that $A[n]$ is the underlying constraint system in our examples and applications.

Other examples of constraint systems include: Rational intervals, Enumerated type, the Kahn constraint system and the Gentzen constraint system (see [SRP91] and [Sar93] for details).

### 3.2 Process Syntax and Notation

In this section we define the formal syntax of ntcc processes. For every process operator, we give an intuitive description of it, which is a short summary of that given in the previous chapter, followed by the syntactic conventions and notations associated to it.

Recall that $\mathcal{C}$ and $\mathcal{V}$ denote the set of constraints modulo logical equivalence and the set of variables, respectively, in the underlying constraint system.

**Definition 3.2.1 (Syntax).** Processes $P$, $Q$, $\ldots \in \text{Proc}$ are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system by the following syntax.

\[
P, Q, \ldots ::= \text{tell}(c) \mid \sum_{i \in I} \text{when } c_i \text{ do } P_i \mid P \parallel Q \mid (\text{local } x) P \\
\quad \mid \text{next } P \mid \text{unless } c \text{ next } P \mid \ast P \mid ! P
\]

The only move or action of a process $\text{tell}(c)$ is to add the constraint $c$ to the current store, thus making $c$ available to other processes in the current time interval.

The guarded-choice $\sum_{i \in I} \text{when } c_i \text{ do }$, where $I$ is a finite set of indexes, represents a process that, in the current time interval, must non-deterministically choose one of the $P_j$ ($j \in I$) whose corresponding guard constraint $c_j$ is entailed by the store. The chosen alternative, if any, precludes the others. If no choice is possible then the summation is precluded from execution in future time interval.

We shall use $N, M$ to stand for summations. We shall often write $\text{when } c_{i_1} \text{ do } P_{i_1} + \ldots + \text{when } c_{i_n} \text{ do } P_{i_n}$, the order of the terms being insignificant, if $I = \{i_1, \ldots, i_n\}$ and, if no ambiguity arises, omit the “\text{when } c \text{ do}” when $c = \text{true}$. The “blind-choice” process $\sum_{i \in I} \text{when } \text{true } \text{do } P_i$, for example, can be written as $\sum_{i \in I} P_i$. We shall omit the “$\sum_{i \in I}$” when $I$ is a singleton. We use $\text{skip}$ as an abbreviation of the empty summation $\sum_{i \in \emptyset} P_i$.

The process $P \parallel Q$ represents the parallel composition of $P$ and $Q$. In one time unit $P$ and $Q$ operate concurrently, communicating through the store.

We adopt the convention that the operator $\parallel$ is left associative. For example, $P_1 \parallel P_2 \parallel P_3$ means $(P_1 \parallel P_2) \parallel P_3$. We shall use $\prod_{i \in I} P_i$, where $I = \{i_1, \ldots, i_n\}$, to denote the parallel composition $P_{i_1} \parallel P_{i_2} \parallel \ldots \parallel P_{i_n}$.
The process $$(\text{local } x) P$$ behaves like $P$, except that all the information about $x$ produced by $P$ can only be seen by $P$ and the information about $x$ produced by other processes cannot be seen by $P$.

We say that $$(\text{local } x) P$$ binds $x$ in $P$. Given a process $Q$, we define in the standard way its bound variables $\text{bv}(Q)$ as the set of variables with a bound occurrence in $Q$, and its free variables $\text{fv}(Q)$ as the set of variables with a non-bound occurrence in $Q$. We use $$(\text{local } x_1 x_2 \ldots x_n) P$$ as an abbreviation of $$(\text{local } x_1) ((\text{local } x_2) \ldots ((\text{local } x_n) P) \ldots))$$.

The only move of next $P$ is an unit-delay for the activation of $P$. The process $P$ will then be activated in the next time interval in some store which may be unrelated to the one in the current time interval. unless $c$ next $P$ is similar, but $P$ will be activated only if $c$ cannot be eventually inferred from the information in the store during the current time interval.

We shall use $\text{next}^n(P)$ to abbreviate $\text{next}(\text{next}(\ldots(\text{next} P)\ldots))$, where $\text{next}$ is repeated $n$ times.

The operator “$*$” corresponds to the unbounded but finite delay operator $\epsilon$ for synchronous CCS [Mil92] and it allows us to express asynchronous behavior across the time intervals. Intuitively, the process $*P$ represents an unbounded summation $P + \text{next} P + \text{next}^2 P + \ldots$, i.e., an arbitrary long but finite delay for the activation of $P$.

The operator “!!” is a delayed version of the replication operator for the $\pi$-calculus [Mil99]: $! P$ represents $P \parallel \text{next} P \parallel \text{next}^2 P \parallel \ldots$, i.e. unboundedly many copies of $P$ but one at a time. The replication operator is the only way of defining infinite behavior across time intervals.

To avoid too many parentheses we adopt the following convention on the binding power of the operators. The replication, star, next, unless and local operators bind the tightest, and parallel composition binds tighter than summation. Thus, for example

$$P + (\text{local } x) \text{next } Q \parallel R \text{ means } P + (((\text{local } x) (\text{next } Q)) \parallel R)$$

### 3.3 Transitions

The operational semantics of ntcc considers transitions (or reduction) between process-store configurations of the form $<P, c>$. We use $\Gamma$ to denote the set of all configurations and $\gamma, \gamma', \ldots$ to range over its elements. Recall that the store $c$ is represented as a constraint in $C$.

**Convention 3.3.1.** Following standard lines (e.g., [dBPP95b]), we extend the syntax of processes with a construct $(\text{local } x, d) P$, which represents the evolution of a process of the form $(\text{local } x) Q$, where $d$ is the local information (or store) produced during this evolution. Initially $d$ is “empty”, so we regard $(\text{local } x) P$ as $(\text{local } x, \text{true}) P$. This syntactic extension is done for operational reasons only.
3.3.1 Structural Congruence

As usual to make the transition system simpler, we wish to identify some process expressions by using a congruence relation \( \cong \) on \( \text{Proc} \). Intuitively \( \cong \) describes irrelevant syntactic aspects of processes.

Let us define precisely what is meant by the term “congruence” of processes as we shall see a few congruence relations in the rest of this thesis. First, we need to introduce the standard notion of process context. Informally speaking, a process context is a process expression with a single hole, represented by \( [\cdot] \), such that placing a process in the hole yields a well-formed process. More precisely,

**Definition 3.3.2 (Process Context).** Process contexts \( C \) are given by the syntax

\[
C ::= [\cdot] \mid \text{when } c \text{ do } C + M \mid C \parallel P \parallel C \parallel C \mid (\text{local } x)C \\
\mid \text{next } C \parallel \text{unless } c \text{ next } C \parallel \ast C \parallel !C
\]

(Recall that \( M \) stands for summations). The process \( C[Q] \) results from the textual substitution of the hole \( [\cdot] \) in \( C \) with \( Q \).

Intuitively, an equivalence relation is a congruence if it equates processes which no context can tell apart. The notion of congruence plays a key role in the theory of processes.

**Definition 3.3.3 (Process Congruence).** An equivalence relation \( \cong \) on processes is said to be a process congruence iff for all contexts \( C \), \( P \cong Q \) implies \( C[P] \cong C[Q] \).

We can now introduce the structural congruence \( \cong \). The definition basically states that \( (\text{Proc}, \cong, \parallel, \text{skip}) \) is a commutative monoid.

**Definition 3.3.4 (Structural Congruence).** Let \( \cong \) be the smallest congruence over processes satisfying the following axioms:

1. \( P \parallel \text{skip} \cong P \)
2. \( P \parallel Q \cong Q \parallel P \)
3. \( P \parallel (Q \parallel R) \cong (P \parallel Q) \parallel R \).

Furthermore, we extend \( \cong \) to configurations by decreeing that \( \langle P, c \rangle \cong \langle Q, c \rangle \) iff \( P \cong Q \).

Some readers may feel that we should have defined \( \langle P, c \rangle \equiv \langle Q, d \rangle \) to hold iff \( P \equiv Q \) and \( c \approx d \), where \( \approx \) denotes logical equivalence. However, remember that we are considering constraints modulo \( \approx \), so such an alternative definition is equivalent to that of Definition 3.3.4.

The reader familiar with the \( \pi \)-calculus may have noticed that in our definition of the structural congruence we excluded the axioms for replication and locality. It turns out that proving decidability of the structural congruence with such axioms is by no means a trivial matter. In fact, it is still an open
question whether or not the standard structural congruence for the \( \pi \)-calculus is decidable. In contrast, our structural congruence is obviously decidable and it will give us the level of simplicity we want for our transition system.

It will be convenient to give a standard form in which processes can be displayed. Recall that \( \prod_{i \in I} P_i \) where \( I = \{ i_1, \ldots, i_n \} \), denotes the parallel composition \( \ldots ((P_{i_1} \parallel P_{i_2}) \parallel P_{i_3}) \ldots ) \parallel P_{i_n} \).

**Definition 3.3.5.** A process \( P = \prod_{i \in I} E_i \) is in standard form if each \( E_i \) takes one of the following forms:

\[
\text{tell}(c), \sum_{j \in J} \text{when } c_j \text{ do } P_j, (\text{local } x) Q, \text{next } Q, \text{unless } c \text{ next } Q, \star Q, !Q
\]

with \( |J| > 0 \) and each \( P_j \) and \( Q \) being themselves in standard form. (If \( I = \emptyset \) we take the standard-form to be \textbf{skip}).

We therefore say that the processes

\[
\text{tell}(c) \quad \text{and} \quad \text{when } c \text{ do } \textbf{skip} + \text{ when } c \text{ do } \text{tell}(d)
\]

are in standard form while the processes

\[
\textbf{skip} \parallel \text{tell}(c) \quad \text{and} \quad (\text{tell}(c) \parallel \text{tell}(d)) \parallel (\text{tell}(c') \parallel \text{tell}(d'))
\]

are not in standard form. Nevertheless, as stated in the proposition below, we can bring every process to a standard form by repeatedly applying the axioms of \( \equiv \). For example, \( \textbf{skip} \parallel \text{tell}(c) \equiv \text{tell}(c) \parallel \textbf{skip} \) by Axiom (2) and then \( \text{tell}(c) \parallel \textbf{skip} \equiv \text{tell}(c) \) by Axiom (1).

**Proposition 3.3.6.** Every process \( P \) is structurally congruent to a process in standard form.

**Proof.** The proof proceeds by induction on the structure of \( P \).

The decidability of \( \equiv \) can be proven by using a simple graph representation of processes in standard form.

**Proposition 3.3.7.** Relation \( \equiv \) is decidable.

**Proof.** The proof is obtained by a reduction to graph isomorphism (which is obviously decidable). Given an arbitrary \( P \) in standard-form we define a node-labeled graph (tree) \( G_P \) which resembles the abstract syntax of \( P \).

The graph for \( P = \text{tell}(c) \) has only one node. Such a node is the root of \( G_P \) and is labeled with \( \text{tell}(c) \). The case \( P = \textbf{skip} \) is similar. The graph for the case \( P = \prod_{i \in I} P_i \), with \( |I| > 0 \), has its root labeled with symbol \( \Pi \).

Its edges are those from \( \Pi \) to each \( G_{P_i} \)'s root where \( i \in I \). The graph for \( P = \sum_{i \in I} \text{when } c_i \text{ do } P_i \) has its root labeled with symbol \( \Sigma \). Its edges are those from \( \Sigma \) to (a node labeled with) \( c_i \) and from \( c_i \) to \( G_{P_i} \)'s root where \( i \in I \).

The other cases are defined similarly.

One can verify by using structural induction that for any two \( P \) and \( Q \) in standard form: \( P \equiv Q \) iff \( G_P \) and \( G_Q \) are isomorphic. The result then follows from Proposition 3.3.6.
3.3.2 Reduction Relations

The operational semantics is given in terms of the reduction relations \( \longrightarrow \) and \( \Rightarrow \) given by the rules 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 \( d' \)”. We then say that \( P' \) is an internal evolution of \( P \).

The observable transition

\[
P \xrightarrow{(c,d)} R
\]

should be read as “\( P \) on input \( c \) from the environment, reduces in one time unit to \( R \) and outputs \( d \) to the environment”. We then say that \( R \) is an (observable) evolution of \( P \).

Intuitively, the above observable reduction is obtained from a sequence of internal reductions starting in \( P \) with initial store \( c \) and terminating in a process \( Q \) with final store \( d \). The process \( R \), which is the one to be executed in the next time interval (or time unit), is obtained by removing from \( Q \) what was meant to be executed only during the current time interval. The store \( d \) is not automatically transferred to the next time interval. Information in \( d \) can only be transferred to the next time unit by \( P \) itself.

**Definition 3.3.8.** Relations \( \longrightarrow \) and \( \Rightarrow \) are the least relations satisfying the rules in Table 3.1.

We now explain the various transition rules in Table 3.1.

**Rules for Internal Transitions**

Let us begin with the rule specifying the reductions of tell processes

\[
\text{TELL} \quad \langle \text{tell}(c), d \rangle \longrightarrow \langle \text{skip}, d \land c \rangle \quad (3.1)
\]

TELL says that a tell process reduces to \( \text{skip} \) while adding its constraint to the store. The rule for summation is given by

\[
\text{SUM} \quad \langle \sum_{i \in I} \text{when } c_i \text{ do } P_i, d \rangle \longrightarrow \langle P_j, d \rangle \quad (3.2)
\]

SUM chooses one of the processes whose corresponding guard is entailed by the store. The following is the standard interleaving rule for parallel composition

\[
\text{PAR} \quad \langle P, c \rangle \longrightarrow \langle P', d \rangle \quad \langle P \parallel Q, c \rangle \longrightarrow \langle P', \parallel Q, d \rangle \quad (3.3)
\]

PAR says that reduction can occur underneath parallel composition. We next have the rule for the conditional unit-delay operator

\[
\text{UNL} \quad \langle \text{unless } c \text{ next } P, d \rangle \longrightarrow \langle \text{skip}, d \rangle \quad (3.4)
\]
Table 3.1: Rules for the internal reduction $\rightarrow$ (upper part) and the observable reduction $\Rightarrow$ (lower part). Notation $\gamma \rightarrow^* \gamma'$ in Rule OBS means that there is no $\gamma'$ such that $\gamma \rightarrow \gamma'$. The relation $\equiv$ and function $F$ are given in Definitions 3.3.4 and 3.3.10, respectively.

UNL says that the execution of process $P$ is precluded if $c$ is entailed by the current store. At this point the reader may wonder about the rule for the case in which unless $c$ next $P$ actually triggers $P$ in the next time unit. This case is similar to the case of next $P$ and they will be clarified with the description of the observable transitions in the next section.

The rule for locality is defined as

$$
\text{LOC } \frac{\langle P, c \land \exists_x d \rangle \rightarrow \langle P', c' \rangle}{\langle \text{local} \ x, c \rangle \ P, d \rightarrow \langle \text{local} \ x, c' \rangle \ P', d \land \exists_x d'}
$$

(3.5)

We shall dwell a little upon the description of Rule LOC as it may seem somewhat complex. Let us consider the process

$$
Q = \langle \text{local} \ x, c \rangle \ P
$$
in Rule 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 \( x \), possibly appearing in the “global” store \( d \), cannot be observed. Thus, before reducing \( P \) we should first hide the information about \( x \) that \( Q \) may have in \( d \). We can do this by existentially quantifying \( x \) in \( d \). Similarly, from the external point of view, the observable information about \( x \) that the reduction of internal agent \( P \) may produce (i.e., \( c' \)) cannot be observed. Thus we hide it by existentially quantifying \( x \) in \( c' \) before adding it to the global store corresponding to the evolution of \( Q \). Additionally, we should make \( c' \) the new private store of the evolution of the internal process for its future reductions.

Let us clarify the hiding behavior by means of an example.

**Example 3.3.9.** Suppose that the underlying constraint system is \( A[n] \). We obtain the following reduction of

\[
Q = (\text{local} \ x, c) \ \text{when} \ (0 < x \land x = y) \ \text{do} \ P',
\]

where the local store is \( c = (x = y) \) and the global store is \( d = (x < y) \).

\[
\frac{\langle \text{when} \ (0 < x \land x = y) \ \text{do} \ P', c \cup \exists x d \rangle \rightarrow \langle P', c \cup \exists x d \rangle}{\langle Q, d \rangle \rightarrow \langle (\text{local} \ x, c \cup \exists x d) \ P', d \land \exists x (c \cup \exists x d) \rangle}
\]

Notice that the \( x \) in \( d \) is hidden, by using existential quantification, in the reduction obtained by using Rule SUM. This expresses that the \( x \) in \( d \) is different from the one bound by the local process — otherwise an inconsistency would be generated (i.e., \( c \land d = \text{false} \)). Rule SUM can be applied since \( c \cup \exists x d \models 0 < y \land x = y \). Observe that the free \( x \) in \( c \cup \exists x d \) is hidden in the global store to indicate that it is different from the global \( x \). Notice, however, that in this case the final store after the reduction is simply \( d \) since \( d = d \land \exists x (c \cup \exists x d) \) (recall that we consider constraint modulo logical equivalence).

We now return to describing the rules. The rule for unbounded but finite delay is defined as:

\[
\text{STAR} \quad \langle \ast P, d \rangle \rightarrow \langle \text{next}^n P, d \rangle \quad \text{if } n \geq 0
\]

(3.6)

Rule STAR says that \( \ast P \) triggers \( P \) in some time interval (either in the current or in a future one). Notice that this rule generates unbounded nondeterminism (or infinite branching) as an arbitrary \( n \geq 0 \) can be chosen for the reduction.

The rule for infinite behavior is given by

\[
\text{REP} \quad \langle ! P, d \rangle \rightarrow \langle P \parallel \text{next} ! P, d \rangle
\]

(3.7)

Rule REP specifies that the process \( ! P \) produces a copy \( P \) at the current time unit, and then persists in the next time unit. Since we delay one time unit
the only way of specifying infinite behavior, there is no risk of infinite behavior within a time unit. Finally, the rule that allows us to make use of the structural congruence is defined as

\[
\begin{align*}
\text{STR} & \quad \gamma_1 \rightarrow \gamma_2 & & \text{if } \gamma_1 \equiv \gamma'_1 \text{ and } \gamma_2 \equiv \gamma'_2 \\
\gamma'_1 & \rightarrow \gamma'_2
\end{align*}
\] (3.8)

Rule STR simply says that structurally congruent configurations have the same reductions.

**Rule for Observable Transitions**

The following is the only rule specifying observable reductions.

\[
\begin{align*}
\text{OBS} & \quad \langle P, c \rangle \rightarrow^* \langle Q, d \rangle \not\rightarrow P \rightarrow^{(c,d)} R & & \text{if } R \equiv F(Q) \\
\text{OBS} & \quad \langle P, c \rangle \rightarrow^* \langle Q, d \rangle \not\rightarrow P \rightarrow^{(c,d)} R
\end{align*}
\] (3.9)

Rule OBS says that an observable transition from \( P \) labeled by \( (c, d) \) is obtained by performing a sequence of internal transitions from the initial configuration \( \langle P, c \rangle \) to a final configuration \( \langle Q, d \rangle \) in which no further internal evolution is possible. The residual process \( R \) to be executed in the next time interval is equivalent to \( F(Q) \) (the “future” of \( Q \)). The process \( F(Q) \), defined below, is obtained by removing from \( Q \) summations that did not trigger activity within the current time interval and any local information which has been stored in \( Q \), and by “unfolding” the sub-terms within “next” and “unless” expressions. This “unfolding” specifies the evolution across time intervals of processes of the form \( \text{next } R \) and \( \text{unless } c \text{ next } R \).

**Definition 3.3.10 (Future Function).** Let \( F : \text{Proc} \rightarrow \text{Proc} \) be the partial function defined by

\[
F(Q) = \begin{cases} 
\text{skip} & \text{if } Q = \sum_{i \in I} \text{ when } c_i \text{ do } Q_i \\
F(Q_1) \parallel F(Q_2) & \text{if } Q = Q_1 \parallel Q_2 \\
(\text{local } x) F(R) & \text{if } Q = (\text{local } x, c) R \\
R & \text{if } Q = \text{next } R \text{ or } Q = \text{unless } c \text{ next } R
\end{cases}
\]

**Remark 3.1.** The function \( F \) does not need to be total since whenever we need to apply \( F \) to a \( Q \) (Rule OBS in Table 3.1), all \( \text{tell}(c) \), \( \star R \) and \( ! R \) in \( Q \) will occur within a “next” or “unless” expression. For simplicity, however, we shall take the liberty of taking \( F(P) = \text{skip} \) if \( P \) is not in the domain of \( F \).

### 3.3.3 Reduction Example

We now give a simple example illustrating a sequence of observable transitions.

**Example 3.3.11.** In Example 2.2.8, we defined a process \( R \) which repeatedly checks the state of \( \text{motor}_1 \) and, whenever a malfunction is reported, tells that
motor₁ must be turned off. We also defined a process \( S \) stating that motor \( \text{motor}_1 \) was doomed to malfunction. Processes \( R \) and \( S \) were defined as:

\[
R = \exists \text{when \( c \) do tell}(c) \\
S = \exists \text{tell}(c)
\]

where \( c = \text{malfunction(\( \text{motor}_1 \),status)} \) and \( e = (\text{motor}_1\_speed = 0) \). Let \( P \) be defined as the parallel composition \( R \parallel S \). Also, let \( S' = \text{tell}(c) \) and \( R' = \exists \text{when \( c \) do tell}(e) \).

One can verify that for an arbitrary \( m > 0 \) the following is a valid sequence of observable transitions starting with \( P \):

\[
\begin{align*}
R \parallel S & \xrightarrow{(c, c \land e)} R \parallel \text{next}^m S' \xrightarrow{\text{true,true}} R \parallel \text{next}^{m-1} S' \xrightarrow{\text{true,true}} \ldots \\
\ldots & \xrightarrow{\text{true,true}} R \parallel S' \xrightarrow{(\text{true},c \land e)} R \xrightarrow{(\text{true,true})} \ldots 
\end{align*}
\]

Intuitively, in the first time interval the environment tells \( c \) (i.e., \( c \) is given as input to \( P \)) thus \( R' \), which is created by \( !R \), tells \( e \). The output is then \( c \land e \). Furthermore, \( S \) creates an \( S' \) which is to be triggered in an arbitrary number of time units \( m + 1 \). In the following time units the environment does not provide any input whatsoever. In the \( m + 1 \)-th time unit \( S' \) tells \( c \) and then \( R' \) tells \( e \).

Let us describe how to obtain the observable reduction

\[
R \parallel S \xrightarrow{(c,c \land e)} R \parallel \text{next}^m S'.
\]

We begin with \( (R \parallel S, c) \longrightarrow \langle (R' \parallel \text{next} R) \parallel S, c \rangle \) which can be obtained from the following derivation:

\[
\begin{align*}
\langle R, c \rangle & \rightarrow \langle R' \parallel \text{next} R, c \rangle \quad \text{REP} \\
\langle R \parallel S, c \rangle & \rightarrow \langle (R' \parallel \text{next} R) \parallel S, c \rangle \quad \text{PAR}
\end{align*}
\]

We then obtain \( \langle (R' \parallel \text{next} R) \parallel S, c \rangle \longrightarrow \langle (\text{tell}(e) \parallel \text{next} R) \parallel S, c \rangle \) as follows.

\[
\begin{align*}
\langle R', c \rangle & \rightarrow \langle \text{tell}(e), c \rangle \quad \text{SUM} \\
\langle R' \parallel \text{next} R, c \rangle & \rightarrow \langle \text{tell}(e) \parallel \text{next} R, c \rangle \quad \text{PAR} \\
\langle (R' \parallel \text{next} R) \parallel S, c \rangle & \rightarrow \langle (\text{tell}(e) \parallel \text{next} R) \parallel S, c \rangle \quad \text{PAR}
\end{align*}
\]

Next we have \( \langle (\text{tell}(e) \parallel \text{next} R) \parallel S, c \rangle \longrightarrow \langle \text{next} R \parallel S, c \land e \rangle \) given by:

\[
\begin{align*}
\langle \text{tell}(e), c \rangle & \rightarrow \langle \text{skip}, c \land e \rangle \quad \text{TELL} \\
\langle \text{tell}(e) \parallel \text{next} R, c \rangle & \rightarrow \langle \text{skip} \parallel \text{next} R, c \land e \rangle \quad \text{PAR} \\
\langle (\text{tell}(e) \parallel \text{next} R) \parallel S, c \rangle & \rightarrow \langle (\text{skip} \parallel \text{next} R) \parallel S, c \land e \rangle \quad \text{PAR} \\
\langle (\text{tell}(e) \parallel \text{next} R) \parallel S, c \rangle & \rightarrow \langle \text{next} R \parallel S, c \land e \rangle \quad \text{STR}
\end{align*}
\]

Finally, we derive \( \langle \text{next} R \parallel S, c \land e \rangle \longrightarrow \langle \text{next} R \parallel \text{next}^{m+1} S', c \land e \rangle \) by using the commutativity of \( \equiv \) (in Rule \text{STR}), \text{PAR} and \text{STAR} as described in
the derivation below.

\[
\begin{align*}
\langle S, c \land e \rangle & \rightarrow \langle \text{next}^{m+1} S', c \land e \rangle & \text{STAR} \\
\langle S \parallel \text{next} R, c \land e \rangle & \rightarrow \langle \text{next}^{m+1} S' \parallel \text{next} R, c \land e \rangle & \text{PAR} \\
\langle \text{next} R \parallel S, c \land e \rangle & \rightarrow \langle \text{next} R \parallel \text{next}^{m+1} S', c \land e \rangle & \text{STR}
\end{align*}
\]

Notice that \( F(\text{next} R \parallel \text{next}^{m+1} S') \equiv R \parallel \text{next}^{m} S' \). Hence,

\[
\frac{\langle R \parallel S, c \rangle \rightarrow^{*} \langle \text{next} R \parallel \text{next}^{m+1} S', c \land e \rangle \rightarrow}{R \parallel S \xrightarrow{[c \land e]} R \parallel \text{next}^{m} S'}
\]

thus deriving the observable transition we wanted to obtain. \( \square \)

### 3.4 Basic Reduction Properties

Here we consider some simple but fundamental properties of the transitions. We shall begin with the properties of the internal reductions and then extend them to the observable ones.

Intuitively, the first property states that, in internal transitions, the store can only be augmented. We can think of this property as an extensiveness property.

**Property 3.4.1 (Internal Extensiveness).** If \( \langle P, c \rangle \rightarrow \langle Q, d \rangle \) then \( d \models c \).

**Proof.** The proof proceeds by a simple induction on (the depth) of the inference of \( \langle P, c \rangle \rightarrow \langle Q, d \rangle \). \( \square \)

Augmenting the store may increase the potentiality of internal reduction, 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 \( c \) is an augmentation of \( d \), provided that \( c \) is “weaker” than \( e \).

**Property 3.4.2 (Internal Potentiality).** If \( e \models c \models d \) and \( \langle P, d \rangle \rightarrow \langle Q, e \rangle \) then \( \langle P, c \rangle \rightarrow \langle Q, e \rangle \).

**Proof.** Assume that \( e \models c \models d \). We proceed by induction on (the depth of) the inference of \( \langle P, d \rangle \rightarrow \langle Q, e \rangle \). We consider the possible cases for the last step of the inference. We confine ourselves to considering three cases.

**Using TELL.** Then \( P \) is \texttt{tell}(c') and \( Q \) is \texttt{skip}, with \( e = d \land c' \). The transition \( \langle \texttt{tell}(c'), c \rangle \rightarrow \langle \texttt{skip}, c \land c' \rangle \) follows from TELL. It follows from the assumption that \( e = d \land c \models c \) and \( c \models d \). Thus, \( (c \land c') = (d \land c') \).

**Using PAR.** Then \( P = P_1 \parallel P_2 \) and \( Q = Q_1 \parallel P_2 \). with \( \langle P_1, d \rangle \rightarrow \langle Q_1, e \rangle \) by a shorter inference. By appeal to induction we have \( \langle P_1, c \rangle \rightarrow \langle Q_1, e \rangle \) and hence \( \langle P, c \rangle \rightarrow \langle Q, e \rangle \) by PAR.

**Using SUM.** Then \( P = \sum_{i \in I} \text{	exttt{when} } c_i \text{ do } P_i \) and \( Q = P_j \) for some \( j \in I \) whence \( e \models c_j \) and \( d = e \). From the assumption \( e \models c \models d \models c_j \) and \( d = e \), thus \( \langle P, c \rangle \rightarrow \langle Q, e \rangle \) follows from SUM.
3.4. Basic Reduction Properties

The two previous statements give us a property which resembles a fixed-point property.

**Property 3.4.3 (Internal Restartability).** Whenever \( \langle P, c \rangle \rightarrow \langle Q, d \rangle \), \( \langle P, d \rangle \rightarrow \langle Q, d \rangle \).

**Proof.** Suppose \( \langle P, c \rangle \rightarrow \langle Q, d \rangle \) then from Property 3.4.1 \( d \models c \). The result follows from Property 3.4.2. \( \square \)

We already pointed out that Rule STAR generates an unbounded branching in the sense that given \( \ast P \) and \( c \) there are infinitely many \( Q \) such that \( \langle \ast P, c \rangle \rightarrow \langle Q, c \rangle \). Nevertheless, if we confine our attention to star-free processes (i.e., processes without occurrences of the \( \ast \) operator) modulo \( \equiv \) then the branching is finite. This property is often referred to as Image-finiteness or finite nondeterminism.

**Property 3.4.4 (Internal Image-Finiteness).** Given a configuration \( \gamma = \langle P, d \rangle \), where \( P \) is a star-free process, the set \( D(\gamma) = \{ \gamma' \mid \gamma \rightarrow \gamma' \} / \equiv \) is finite.

**Proof.** Let \( \gamma = \langle P, d \rangle \) be an arbitrary configuration. The proof proceeds by induction on the structure of \( P \). Here we show some cases.

\[
P = \text{tell}(c).
\]

If \( \langle P, d \rangle \rightarrow \langle Q, e \rangle \) then it must be the case that \( Q \equiv \text{skip} \) and \( e = d \land c \). Hence \( |D(\gamma)| = 1 \).

\[
P = \sum_{i \in I} \text{when } c_i \text{ do } P_i.
\]

If \( \langle P, d \rangle \rightarrow \langle Q, e \rangle \) then one can verify that \( Q \equiv P_j \) for some \( j \in I \) and \( e = d \). Thus, \( |D(\gamma)| \leq |I| \) is finite since \( I \) is finite.

\[
P = P_1 \parallel P_2.
\]

Suppose that \( \langle P, d \rangle \rightarrow \langle Q, e \rangle \). Then either \( Q, e \equiv \langle P'_1 \parallel P_2, e \rangle \) or \( Q, e \equiv \langle P'_1 \parallel P_2, e \rangle \) where \( \gamma_i = \langle P_i, d \rangle \rightarrow \langle P'_i, e \rangle \) for \( i \in \{1, 2\} \). By appeal to induction we conclude that \( |D(\gamma)| \leq |D(\gamma_1)| \times |D(\gamma_2)| \) is finite.

\[
P = (\text{local } x, c) R.
\]

Suppose that \( \langle P, d \rangle \rightarrow \langle Q, e \rangle \). We must therefore have \( Q, d \equiv \langle (\text{local } x, c') R', d \land \exists x c' \rangle \) where \( \gamma_1 = \langle R, c \land \exists x d \rangle \rightarrow \langle R', c' \rangle \). By appeal to induction we conclude that \( |D(\gamma)| = |D(\gamma_1)| \) is finite. \( \square \)

We have mentioned that the internal behavior is always finite as a result of the delayed nature of our replication operator (Rule REP). The following definition will give us an upper bound on the number of internal actions within a time unit of an arbitrary process.

**Definition 3.4.5 (Function \( \text{na}(\cdot) \)).** Let \( \text{na} : \text{Proc} \rightarrow \mathcal{N} \) be recursively defined by:

\[
\begin{align*}
\text{na}(\text{skip}) &= \text{na}(\text{unless } c \text{ next } P) = \text{na}(\text{next } P) = 0 \\
\text{na}(\text{tell}(c)) &= \text{na}(\text{unless } c \text{ next } P) = 1 \\
\text{na}(P \parallel Q) &= \text{na}(P) + \text{na}(Q) \\
\text{na}(\sum_{i \in I} \text{when } c_i \text{ do } P_i) &= 1 + \max_{i \in I} \text{na}(P_i) \\
\text{na}(\text{local } x) P &= \text{na}(\ast P) = \text{na}(!P) = 1 + \text{na}(P).
\end{align*}
\]
The following proposition states that function $na(.)$ is invariant wrt structural congruence.

**Proposition 3.4.6.** $P \equiv Q$ then $na(P) = na(Q)$.

**Proof.** Every application of an axiom in Definition 3.3.4 preserves $na(.)$. 

The following lemma states that every sequence of internal transitions is indeed terminating. The proof of this lemma also states that $na(P)$ is indeed an upper bound on the number of internal transitions (actions) starting from $P$.

**Lemma 3.4.7.** Every sequence of internal transitions is terminating, i.e., there are no infinite sequences

$$
\langle P_1, d_1 \rangle \rightarrow \langle P_2, d_2 \rangle \rightarrow \ldots
$$

**Proof.** It enough to prove that if $\langle P, d \rangle \rightarrow \langle Q, e \rangle$ then $na(P) > na(Q)$. The proof proceeds by induction on (the depth of) the inference of $\langle P, d \rangle \rightarrow \langle Q, e \rangle$ with the help of Proposition 3.4.6. Here we illustrate some cases. The key case is, of course, the rule for infinite behavior REP.

**Using TELL.** Then $P = \text{tell}(e')$ and $Q = \text{skip}$. We are done since we have $na(\text{tell}(e')) = 1 > na(\text{skip}) = 0$.

**Using PAR.** Then $P = (P_1 \parallel P_2)$ and $Q = (Q_1 \parallel P_2)$ with $\langle P_1, d \rangle \rightarrow \langle Q_1, e \rangle$ by a shorter inference. We have $na(P) = na(P_1) + na(P_2)$ and also $na(Q) = na(Q_1) + na(P_2)$. By appeal to induction $na(P_1) > na(Q_1)$, hence $na(P) > na(Q)$.

**Using SUM.** Then $P = \sum_{i \in I} \text{when } c_i \text{ do } P_i$ and $Q = P_j$ for some $j \in I$. We have $na(P) = (1 + \max_{i \in I} na(P_i)) > na(P_j)$ for every $j \in I$.

**Using STR.** We have $P \equiv P'$ and $Q \equiv Q'$ with $\langle P', d \rangle \rightarrow \langle Q', e \rangle$ by a shorter inference. By using Proposition 3.4.6 and the induction we can conclude that $na(P) = na(P') > na(Q') = na(Q)$.

**Using REP.** Then $P = !R$ and $Q = R \parallel \text{next} !R$. We are done since we have $na(P) = 1 + na(R) > n(R) = na(Q)$.

The above properties of the internal relation $\rightarrow$ extend to the observable relation $\rightarrow \rightarrow$.

**Theorem 3.4.8.** Relation $\rightarrow \rightarrow$ satisfies the following:

1. *(Extensiveness)* If $P \xrightarrow{(c,d)} Q$ then $d \models c$.

2. *(Potentiality)* If $e \models c \models d$ and $P \xrightarrow{(d,e)} Q$ then $P \xrightarrow{(c,e)} Q$.

3. *(Restartability)* If $P \xrightarrow{(c,d)} Q$ then $P \xrightarrow{(d,d)} Q$. 

4. (Image-Finiteness) For each star-free process \( P \) and constraint \( c \), the set 
\[ \{(Q, d) \mid P \xrightarrow{(c,d)} Q\} \equiv \] is finite.

Proof. Cases (1-3) can be established by appealing to Properties 3.4.1, 3.4.2, 3.4.3. Consider (1). By definition, if \( P \xrightarrow{(c,d)} Q \) then there is a sequence 
\[ (P_1, c_1) \rightarrow (P_2, c_2) \rightarrow \ldots \rightarrow (P_n, c_n) \]
with \( P = P_1 \), \( Q = F(P_n) \), \( c = c_1 \) and \( d = c_n \). We have \( c_n \models \ldots \models c_2 \models c_1 \) from Property 3.4.1, so \( d \models c \). Cases (2) and (3) are similar.

Case (4) follows from the image-finiteness of the internal reduction (Property 3.4.4) and the finiteness of sequences of internal transitions (Lemma 3.4.7) by a simple application of König’s Lemma \(^1\).

3.5 Observable Behavior

In this section we introduce some notions of what an observer can see from a process behavior. We shall refer to such notions as process observations. We assume that what happens within a time unit cannot be directly observed, and thus we abstract from internal transitions. The \( \text{ntcc} \) calculus makes it easy to focus on the observation of input-output events in which a given process engages and the order in which they occur.

Notation 3.5.1. Throughout this dissertation \( \mathcal{C}^\omega \) will denote the set of infinite sequences of constraints in the underlying set of constraints \( \mathcal{C} \). We use \( \alpha, \alpha', \alpha_1, \alpha_2 \ldots \) to range over \( \mathcal{C}^\omega \).

3.5.1 Interpreting Process Runs.

Let us consider the sequence of observable transitions

\[
P = P_1 \xrightarrow{(c_1, c'_1)} P_2 \xrightarrow{(c_2, c'_2)} P_3 \xrightarrow{(c_3, c'_3)} \ldots \tag{3.10}
\]

We shall also represent the above run as \( P \xrightarrow{\langle \alpha, \alpha' \rangle} \omega \) where \( \alpha = c_1.c_2.c_3.\ldots \) and \( \alpha' = c'_1.c'_2.c'_3.\ldots \)

Input-Output Behavior

The run in Equation 3.10 can be interpreted as an interaction between the system \( P \) and the environment. At the time unit \( i \), the environment provides a stimulus \( c_i \) and \( P \) produces \( c'_i \) as a response. As observers, we can see that on input \( \alpha \) the process \( P \) responds with \( \alpha' \). We then regard \( \langle \alpha, \alpha' \rangle \) as a reactive observation of \( P \). Given \( P \) we shall refer to the set of all its reactive observations as the input-output behavior of \( P \).\(^\text{1}\)

\(^1\)König’s Lemma states that every infinite finitely-branching tree has an infinite path.
Default Output Behavior

Alternatively, if $\alpha = \text{true}^\omega$, we can interpret the run as an interaction among the parallel components in $P$ without the influence of any (external) environment – i.e., each component is part of the environment of the others. In this case $\alpha$ can be regarded as being irrelevant and $\alpha'$ can be regarded as a timed observation of such an interaction in $P$. Thus, as observers what we see is that $P$ produces $\alpha'$ in its own right. We shall refer to the set of all timed observations of a process $P$ as the (default) output behavior of $P$.

Quiescent Sequences

Another observation we can make of a process is its quiescent input sequences. Intuitively, those are sequences on input of which $P$ can run without adding any information whatsoever, therefore what we observe is that the input and output sequences coincide, i.e. $\alpha = \alpha'$. More precisely, the quiescent sequences of $P$ are those sequences $\alpha$ such that $P \xrightarrow{(\alpha, \alpha)} \omega$.

It turns out that the set of quiescent sequences of a process $P$ can be characterized as the set of all infinite sequences that $P$ can possibly output under the influence of arbitrary environments. More precisely, the set of all $\alpha$ for which there exists an $\alpha'$ such that $P \xrightarrow{(\alpha', \alpha)} \omega$. The theorem below states the equivalence above claimed.

**Theorem 3.5.2.** For all $P$ and $\alpha$, $P \xrightarrow{(\alpha, \alpha)} \omega$ iff $P \xrightarrow{(\alpha', \alpha)} \omega$ for some $\alpha'$.

**Proof.** The “if” direction follows from Theorem 3.4.8(3), i.e., the Restartability of observable transitions. The other direction is trivial. \[\square\]

Strongest Postcondition

In imperative programming we can think of the strongest-postcondition of a given program (with no precondition on the inputs for the program) simply as the range of the program. That is, the set of all outputs that the program can return on arbitrary inputs. Similarly, in our setting, we can think of the set of all infinite sequences that $P$ can possibly output under arbitrary environments as the strongest-postcondition of $P$. Consequently, justified in the equivalence stated in Theorem 3.5.2, we shall also refer to the set of quiescent sequences of $P$ as the strongest postcondition of $P$, written $\text{sp}(P)$. In fact, in Chapter 5 we adapt to our case Dijkstra’s strongest postcondition approach to program verification. We shall then see that proving whether $P$ satisfies a given property $A$ (over sequences in $\mathcal{C}^\omega$) in the presence of arbitrary environments, reduces to proving whether $\text{sp}(P)$ is included in the set of sequences satisfying $A$.

3.5.2 Summary of Observable Behavior and Equivalences

The following definition states the various notions of observable behavior mentioned above.
Definition 3.5.3 (Observable Behavior). The behavioral observations that can be made of a process are:

1. The input-output (or stimulus-response) behavior of $P$, written $io(P)$, defined as

   $$io(P) = \{(\alpha, \alpha') \mid P \xrightarrow{(\alpha, \alpha') \omega} \}.$$ 

2. The (default) output behavior of $P$, written $o(P)$, defined as

   $$o(P) = \{\alpha' \mid P \xrightarrow{\text{true}_{\alpha'} \omega}\}.$$ 

3. The strongest postcondition behavior of $P$, written $sp(P)$, defined as

   $$sp(P) = \{\alpha \mid P \xrightarrow{(\alpha', \alpha) \omega} \text{ for some } \alpha'\}.$$ 

The following are the obvious equivalences induced by our behavioral observations.

Definition 3.5.4 (Behavioral Equivalences). Define $\sim_{io}$, $\sim_{o}$ and $\sim_{sp}$ as the equivalences over processes given by

$$P \sim_{io} Q \iff io(P) = io(Q),$$

$$P \sim_{o} Q \iff o(P) = o(Q),$$

$$P \sim_{sp} Q \iff sp(P) = sp(Q).$$

We shall refer to equivalences defined above as observational equivalences as they identify processes whose internal behavior may differ widely (e.g. in the number of internal actions). Such an abstraction from internal behavior is essential in the theory of several process calculi; most notably in weak bisimilarity for CCS [Mil89].

Several typical questions about these equivalence may then arise. For example, one may wonder which of them coincides with their corresponding induced congruences and whether there are interesting relationship between them. In the following chapters we shall address these issues.

3.6 Summary and Related Work

In this chapter we described the ntcc calculus formally. We set up the notion of constraint system by using first-order logic. We introduced the finite-domain constraint systems $\texttt{FD}[n]$ and $\texttt{A}[n]$. The former provides a theory of variables ranging over a finite domain of size $n$ with syntactic equality. The latter provides the theory of arithmetic modulo $n$. We shall use $\texttt{FD}[n]$ in the impossibility results involving finite domains and $\texttt{A}[n]$ in our application examples.

A more general formulation of the notion of constraint system is given in [SRP91] by using Scott’s information systems [Sco82] without consistency structure.

We defined an operational semantics based on internal and observable transitions with the help of a decidable structural equivalence. The internal (or
unobservable) transitions specify behavior within time intervals whereas the observable transitions specify behavior across time intervals.

Our definition of the operational semantics follows that of [FGMP97] for untimed ccp in the use of configuration given as process-store pairs. It also follows the reduction semantics of π-calculus in the use of a structural congruence. The operational semantics of tcc [SJG94a] uses multi-sets of processes instead of configurations. Furthermore, in tcc there is no explicit notion of observable and unobservable transitions.

We also introduced some behavioral observations that can be made of processes. The first such an observation, input-output behavior, is the set of infinite sequences of input-output interactions in which a process can engage with an environment. The second, called default output behavior, focuses on the output behavior of processes in the absence of an external environment, i.e., the output sequences on the irrelevant input sequence true. The third, called strongest-postcondition, focuses on the output behavior in the presence of arbitrary environments. We proved that this third observation is equivalent to the observation that focus on the output sequences matching their corresponding input. We called such output sequences quiescent.

The notion of strongest postcondition for ccp is presented in [dBGMP97] for the untimed case and thus is concerned only with terminating processes. By contrast in tcc we will be able to use the strongest postcondition approach for non-terminating processes.

Based upon the observations mentioned above, we introduced different notions of behavioral equivalence that abstract away from internal behavior. In the following chapters, we shall study in more detail these behavioral observations and equivalences. In particular, Chapters 4 and 5 are devoted to the study of a denotational semantics and a temporal logic for the strongest postcondition. Chapter 7 focuses on the relationship among the various equivalences and their corresponding congruences, and their decidability.

The formal operational description of the tcc calculus was originally published in [PV01]. A more detailed description is given in [NPV02b].
Chapter 4

Denotational Semantics

Many people will only be satisfied with the semantic theory of concurrent systems when - eventually - it becomes an abstract theory as well as a formal one.

— Robin Milner

In the previous chapter we introduced the notion of strongest-postcondition of ntcc processes in operational terms. In this chapter we give an abstract denotational model of this notion. Such a model will be of great help when arguing about the strongest-postcondition in the following chapter.

Our denotational model follows ideas developed in the works [dBGMP97] and [SJG94a] for the untimed ccc and tcc case, respectively. The ntcc case, however, presents new technical problems to deal with. One such a problem arises from the presence of nondeterminism: The strongest-postcondition for the hiding operator cannot be specified compositionally (see [dBGMP97]). Another technical problem arises from the combination between hiding and the “unless” operator: The “unless” operator, unlike all other operators of the calculus, is not monotonic. Therefore, our goal is to identify crucial conditions under which the semantics is complete wrt our observables. We shall see that indeed a significant fragment of the calculus satisfies such conditions.

Recall that the strongest postcondition of a process $P$, $\text{sp}(P)$ can be described in two equivalent ways. The first one is by Definition 3.5.3: $\text{sp}(P)$ is equivalent to the set of output sequences of $P$ on input sequences from arbitrary environments. The second one is by Theorem 3.5.2: $\text{sp}(P)$ denotes the set of quiescent sequences of $P$. These are sequences on which $P$ can run without adding any information whatsoever. In this chapter we find it convenient to work with the second description.

4.1 The Semantic Definitions

In this section we describe the denotation semantics of the various process constructs. The semantics is defined as a function $\llbracket \cdot \rrbracket$ which associates to each process a set of infinite sequences $\alpha \in \mathcal{C}^\omega$. The definition of this function is given in Table 4.1.

We shall use the following notation throughout this dissertation.
**Notation 4.1.1.** We use $\exists_x \alpha$ to represent the sequence obtained by applying $\exists_x$ to each constraint of the infinite sequence $\alpha \in C^\omega$. Notation $\alpha(i)$ denotes the $i$-th element of $\alpha$. We shall use $\beta$ to represent elements of $C^\ast$ which denotes the set of finite sequences of constraints in $C$. Furthermore we use $\beta.\alpha$ to represent the concatenation of $\beta$ and $\alpha$.

We shall now give some intuition about the semantic definition of each process. The denotation of $\text{tell}(c)$ is given by

$$\text{DTELL: } \llbracket \text{tell}(c) \rrbracket = \{d.\alpha \mid d \models c\}. \quad (4.1)$$

Equation DTELL expresses that the sequences on which $\text{tell}(c)$ can run without adding information are those whose first element is stronger than $c$. The denotation of the summation operator is defined as

$$\text{DSUM: } \llbracket \sum_{i \in I} \text{when } c_i \text{ do } P_i \rrbracket = \bigcup_{i \in I} \{d.\alpha \mid d \models c_i \text{ and } d.\alpha \in \llbracket P_i \rrbracket\} \quad (4.2)$$

$$\bigcap_{i \in I} \{d.\alpha \mid d \not\models c_i\}.$$

Thus, the sequences on which $\sum_{i \in I} \text{when } c_i \text{ do } P_i$ can run without adding information are those whose first element entails either: one of the $c_i$ and thus they should be also quiescent for $P_i$, or none of the $c_i$. The denotation for parallel composition is given by

$$\text{DPAR: } [P \parallel Q] = [P] \cap [Q]. \quad (4.3)$$

This equation says that a sequence is quiescent for the parallel composition iff it is quiescent for each component. Next we have the denotation of the local operator

$$\text{DLOC: } \llbracket (\text{local } x) P \rrbracket = \{\alpha \mid \text{there exists } \alpha' \in [P] \text{ s.t. } \exists_x \alpha' = \exists_x \alpha\} \quad (4.4)$$

Operationally, every output of $(\text{local } x) P$ on an input $\alpha_{in}$ is taken to be $\exists_x \alpha_{out}$, where $\alpha_{out}$ is an output of $P$ on input $\exists_x \alpha_{in}$. (i.e. we hide, via existential quantification, the $x$ in $\alpha_{in}$ as it is to be different from $P$’s private $x$. For the same reason we also hide the $x$ in $P$’s output $\alpha_{out}$). Therefore, if running $(\text{local } x) P$ on input $\alpha$ adds new information to that of $\alpha$, then that information is certainly not about the $x$ in $\alpha$. Suppose now that $P$ can run on $\alpha'$ without adding any information to it (i.e., $\alpha' \in [P]$) and moreover that such an $\alpha'$ has the same information as $\alpha$ has except for that about $x$ (i.e., $\exists_x \alpha' = \exists_x \alpha$). We should then expect $(\text{local } x) P$ to be able to run on $\alpha$ without adding any information.

The process $\text{next } P$ has influence only in the suffix of the input sequence, thus $d.\alpha$ is quiescent for it if $\alpha$ is quiescent for $P$. This gives us the following semantic equation:

$$\text{DNEXT: } \llbracket \text{next } P \rrbracket = \{d.\alpha \mid \alpha \in \llbracket P \rrbracket\}. \quad (4.5)$$
The process \textbf{unless} \textbf{next} $P$ is similar to \textbf{next} $P$, except that it can add information to the suffix of the input sequence only if the first element does not entail $c$. Thus,

\[
\text{DUNL: } \left[\text{unless } c \text{ next } P\right] = \{d.\alpha \mid d \models c\} \\
\cup \{d.\alpha \mid d \not\models c \text{ and } \alpha \in [P]\}
\]  

(4.6)

The semantic equation for process $! P$ can be derived from that of parallel composition and the next operator. Recall that process $! P$ can be viewed as $P \parallel \text{next} P \parallel \text{next}^2 P \parallel \ldots$, i.e., a copy of $P$ at each time unit. Hence, a sequence is quiescent for $! P$ if every suffix of it is quiescent for $P$. We then have

\[
\text{DREP: } [! P] = \{\alpha \mid \text{for all } \beta, \alpha' \text{ s.t. } \alpha = \beta.\alpha', \text{ we have } \alpha' \in [P]\} \quad \text{(4.7)}
\]

Finally we give the semantic equation for $* P$. The process $* P$ can be viewed as $P + \text{next} P + \text{next}^2 P + \ldots$, i.e., arbitrary delay of $P$. Thus, analogous to the replication case, we can derive its equation from that of summation and the next operator. A sequence is quiescent for $* P$ if there exists a suffix of it which is quiescent for $P$. The other rules can be explained analogously. The semantic equation is therefore

\[
\text{DSTAR: } [* P] = \{\beta.\alpha \mid \alpha \in [P]\} \quad \text{(4.8)}
\]

Remark 4.1 (Fixed Points). Notice that the $!$ and the $*$ operators are dual. In fact, we could define their denotational semantics as follows:

\[
[! P] = \nu_X ([P] \cap \{d.\alpha \mid d \in \mathcal{C}, \alpha \in X\}) \\
[* P] = \mu_X ([P] \cup \{d.\alpha \mid d \in \mathcal{C}, \alpha \in X\})
\]

where $\nu$ and $\mu$ represent respectively the greatest and the least fix-point operators in the complete lattice $(P(\mathcal{C}^\omega), \subseteq)$.

4.2 Soundness

This section is entirely devoted to the proof of the following soundness theorem. This theorem states that the operational notion of $sp(P)$ preserves the denotation.

Theorem 4.2.1 (Soundness). \textit{For every ntcc process $P$, $sp(P) \subseteq [P]$}.

\textbf{Proof.} Let us assume that $\alpha \in sp(P)$. From Theorem 3.5.2, we have $P \xrightarrow{(\alpha,\alpha)} \omega$. The proof proceeds by induction on the structure of $P$.

$P = \text{tell}(c)$. Let $\alpha = d.\alpha'$. We must have

\[
\text{tell}(c) \xrightarrow{(d,d)} Q \xrightarrow{(\alpha',\alpha')} \omega
\]

for some $Q$. But this is possible only if $d \models c$. Hence from the semantic equation DTELL we conclude $\alpha \in [\text{tell}(c)]$. 

\begin{tabular}{|l|l|}
\hline
DTLL: & \( [\text{tell}(c)] = \{d.\alpha \mid d \models c \} \) \\
DSUM: & \[
\{ \sum_{i \in I} \text{when } c_i \text{ do } P_i \} = \bigcup_{i \in I} \{d.\alpha \mid d \models c_i \text{ and } d.\alpha \in [P_i] \} \\
& \bigcup \{d.\alpha \mid d \not\models c_i \} \\
DPAR: & \([P \parallel Q] = [P] \cap [Q] \) \\
DLOC: & \([(\text{local } x) \ P] = \{\alpha \mid \text{there exists } \alpha' \in [P] \text{ s.t. } \exists x.\alpha' = \exists_x \alpha \} \) \\
DNEXT: & \([\text{next } P] = \{d.\alpha \mid \alpha \in [P] \} \) \\
DUNL: & \([\text{unless } c \text{ next } P] = \{d.\alpha \mid d \models c \} \\
& \bigcup \{d.\alpha \mid d \not\models c \text{ and } \alpha \in [P] \} \) \\
DREP: & \([!P] = \{\alpha \mid \text{for all } \beta,\alpha' \text{ s.t. } \alpha = \beta.\alpha', \text{ we have } \alpha' \in [P] \} \) \\
DSTAR: & \([* P] = \{\beta.\alpha \mid \alpha \in [P] \} \) \\
\hline
\end{tabular}

Table 4.1: Denotational semantics of ntcc. The symbols \( \alpha \) and \( \alpha' \) range over the set of infinite sequences of constraints \( C^{\omega} \). The symbol \( \beta \) ranges over the set of finite sequences of constraints \( C^{*} \). Notation \( \exists_x \alpha \) denotes the sequence resulting by applying \( \exists_x \) to each constraint in \( \alpha \).

\[ P = \sum_{i \in I} \text{when } c_i \text{ do } P_i. \] Let \( \alpha = d.\alpha' \). We distinguish two cases:

1. There exists \( i \in I \) such that \( d \models c_i \). In this case we can infer that for some \( R, \langle P, d \rangle \rightarrow \langle P_i, d \rangle, \langle P, d \rangle \rightarrow^{*} \langle R, d \rangle \rightarrow \) and \( F(R) \xrightarrow{\alpha' \alpha'} \omega \). Thus, \( P_i \xrightarrow{(d.\alpha', d.\alpha')} \omega \). By inductive hypothesis, \( d.\alpha' \in [P_i] \) and therefore, from DSUM, \( d.\alpha' \in [P] \).

2. For all \( i \in I, d \not\models c_i \). This case is immediate.

\[ P = Q \parallel R. \] Let \( \alpha = c_1.c_2 \ldots c_n \ldots \). One can then verify that there is a derivation of the form

\[ Q \parallel R = Q_1 \parallel R_1 \xrightarrow{(c_1.c_1)} Q_2 \parallel R_2 \xrightarrow{(c_2.c_2)} \ldots Q_n \parallel R_n \xrightarrow{(c_n.c_n)} \ldots \]

Such that each \( Q_{i+1} \) is an evolution of \( Q_i \) and each \( R_{i+1} \) is an evolution of \( R_i \) \((i > 1)\). From the operational semantics of the parallel operator we derive
\[ Q = Q_1 \xrightarrow{(c_1,c_1)} Q_2 \xrightarrow{(c_2,c_2)} \ldots Q_n \xrightarrow{(c_n,c_n)} \ldots \]

and
\[ R = R_1 \xrightarrow{(c_1,c_1)} R_2 \xrightarrow{(c_2,c_2)} \ldots R_n \xrightarrow{(c_n,c_n)} \ldots \]

Therefore, \( Q \xrightarrow{(\alpha,\alpha)} \omega \) and \( R \xrightarrow{(\alpha,\alpha)} \omega \). By inductive hypothesis, we have \( \alpha \in [Q] \) and \( \alpha \in [R] \). From DPAR we conclude \( \alpha \in [Q \parallel R] \).

\[ P = (\text{local } x) P'. \] Let \( \alpha = c_1.c_2 \ldots c_n \ldots \). We can verify that there must be a derivation of the form
\[
(\text{local } x) P' = (\text{local } x) P'_1 \xrightarrow{(c_1,c_1')} \ldots (\text{local } x) P'_n \xrightarrow{(c_n,c_n')} \ldots
\]

From Rule LOC and induction on the length of each observable transition, we derive
\[
P' = P'_1 \xrightarrow{(\exists_x c_1.c'_1)} P'_2 \xrightarrow{(\exists_x c_2.c'_2)} \ldots P'_n \xrightarrow{(\exists_x c_n.c'_n)} \ldots
\]

where for every \( i \geq 1 \) there exists a \( d_i \) such that \( c'_i = (\exists_x c_i) \land d_i \) and \( c_i = c_i \land \exists_x d_i \). By appealing to Theorem 3.5.2 we obtain
\[
P' = P'_1 \xrightarrow{(c'_1,c'_1')} P'_2 \xrightarrow{(c'_2,c'_2')} \ldots P'_n \xrightarrow{(c'_n,c'_n')} \ldots
\]

and therefore, by the inductive hypothesis, we have \( \alpha' = c'_1.c'_2 \ldots c'_n \ldots \)
in \([P']\]. Finally, observe that \( \exists_x \alpha = \exists_x \alpha' \), since for each \( i \geq 1 \) we have \( \exists_x c_i = \exists_x (c_i \land \exists_x d_i) = (\exists_x c_i) \land (\exists_x d_i) = \exists_x (\exists_x c_i) \land d_i = \exists_x c'_i \). Hence, \( \alpha \in [P] \) by DLOC.

\[ P = \text{next } P'. \] Let \( \alpha = d.\alpha' \). Then we must have

\[ \text{next } P' \xrightarrow{(d,d)} Q \xrightarrow{(\alpha',\alpha')} \omega \]

for some \( Q \). We must have \( Q \equiv F(\text{next } P') = P' \). Then, we derive \( \alpha' \in [P'] \) by appeal to induction, and therefore \( d.\alpha' \in [P] \) from DNEXT.

\[ P = \text{unless } c \text{ next } P'. \] Let \( \alpha = d.\alpha' \). We distinguish two cases:

1. \( d \models c \). Immediate.

2. \( d \not\models c \). This case is similar to the case \( P = \text{next } P' \).
\[ P = !Q. \] Let \( \alpha = c_1 \cdot c_2 \cdot c_3 \ldots c_n \ldots \). We can verify that if \( \langle !Q, c_1 \rangle \rightarrow \langle R, c_1 \rangle \), then \( R \equiv Q \parallel \textbf{next} \! Q \). We then must have \( \langle Q \parallel \textbf{next} \! Q, c_1 \rangle \rightarrow^* \langle Q' \parallel \textbf{next} \! Q, c_1 \rangle \). Since \( F(Q' \parallel \textbf{next} \! Q) = F(Q') \parallel Q \), by repeating this reasoning and that of the parallel case, we can obtain for \( Q = Q_{1,1} \) a derivation of the form:

\[
\begin{align*}
Q_{1,1} & \xrightarrow{(c_1,c_1)} Q_{1,2} \parallel Q_{1,1} \\
& \xrightarrow{(c_2,c_2)} Q_{1,3} \parallel Q_{2,2} \parallel Q_{1,1} \\
& \xrightarrow{(c_3,c_3)} Q_{1,4} \parallel Q_{2,3} \parallel Q_{3,2} \parallel Q_{1,1} \\
& \ldots \\
& \xrightarrow{(c_{n-1},c_{n-1})} Q_{1,n} \parallel Q_{2,n-1} \parallel Q_{3,n-2} \parallel \ldots \parallel Q_{n-1,2} \parallel Q_{1,1} \\
& \xrightarrow{(c_n,c_n)} \ldots
\end{align*}
\]

where each parallel component contributes in the following way:

\[
\begin{align*}
Q_{1,1} & \xrightarrow{(c_1,c_1)} Q_{1,2} \parallel Q_{1,1} \\
& \xrightarrow{(c_2,c_2)} Q_{1,3} \parallel Q_{2,2} \parallel Q_{1,1} \\
& \xrightarrow{(c_3,c_3)} Q_{1,4} \parallel Q_{2,3} \parallel Q_{3,2} \parallel Q_{1,1} \\
& \ldots \\
& \xrightarrow{(c_{n-1},c_{n-1})} Q_{1,n} \parallel Q_{2,n-1} \parallel Q_{3,n-2} \parallel \ldots \\
& \xrightarrow{(c_n,c_n)} \ldots
\end{align*}
\]

By the inductive hypothesis we derive

\[
\begin{align*}
c_1 \cdot c_2 \cdot c_3 \ldots c_n \ldots & \in [Q] \\
c_2 \cdot c_3 \ldots c_n \ldots & \in [Q] \\
c_3 \ldots c_n \ldots & \in [Q] \\
& \ldots
\end{align*}
\]

From DREP we conclude \( c_1 \cdot c_2 \cdot c_3 \ldots c_n \ldots \in [P] \).

\[ P = \ast Q. \] Let \( \alpha = c_1 \cdot c_2 \cdot c_3 \ldots c_n \ldots \). If \( \langle \ast Q, c_1 \rangle \rightarrow \langle R, c_1 \rangle \) then \( R \equiv \textbf{next}^k Q \) for some \( k \geq 0 \). We distinguish two cases.

\( k = 0 \). In this case we have

\[ \langle \ast Q, c_1 \rangle \rightarrow \langle Q, c_1 \rangle \rightarrow^* \langle Q', c_1 \rangle \rightarrow \]

and \( F(Q') \xrightarrow{(\alpha', \alpha')} \omega \), where \( \alpha' = c_2 \cdot c_3 \ldots c_n \ldots \). Hence \( Q \xrightarrow{(\alpha, \alpha)} \omega \).

From the inductive hypothesis we derive \( \alpha \in [Q] \) and therefore, from DSTAR, \( \alpha \in [P] \).

\( k \geq 1 \). Since there are no internal transitions from \( \langle \textbf{next}^k Q, c_1 \rangle \) for \( k \geq 1 \), and \( F(\textbf{next}^i Q) = \textbf{next}^{i-1} Q \) for every \( i \geq 1 \), we must have that

\[
\ast Q \xrightarrow{(c_1,c_1)} \textbf{next}^{k-1} Q \xrightarrow{(c_2,c_2)} \ldots \textbf{next} \ast Q \xrightarrow{(c_k,c_k)} Q
\]

and
\[ Q = Q_1 (c_{k+1},c_{k+1}) \Rightarrow Q_2 (c_{k+2},c_{k+2}) \Rightarrow \ldots Q_{n-k} (c_n,c_n) \Rightarrow \ldots \]

Hence we derive, from the inductive hypothesis, that the sequence \( c_{k+1},c_{k+2},c_{k+3},\ldots,c_n \ldots \) is in \( [Q] \). From DSTAR we conclude \( \alpha \in [P] \).

\[ \Box \]

4.3 Completeness of the Denotation

We now explore the reverse of Theorem 4.2.1, namely completeness, which does not hold in general. The essential reason is the combination of the local operator, which decreases the store by hiding away information, and constructs whose input-output relation is not conversely monotonic in the sense described below. Let us first define the following relation on sequences. Recall \( \alpha(i) \) denotes the \( i \)-th element in \( \alpha \).

**Definition 4.3.1 (Sequence Order).** The (partial) ordering \( \leq \) on \( C^\omega \) is defined by \( \alpha \leq \alpha' \) iff for all \( i \geq 1, \alpha'(i) \models \alpha(i) \) holds.

We shall call a relation \( R \) monotonic if whenever \( (\alpha_1,\alpha_2) \in R \) and \( \alpha_1 \leq \alpha'_1 \) then there exists \( \alpha'_2 \) such that \( (\alpha'_1,\alpha'_2) \in R \) and \( \alpha_2 \leq \alpha'_2 \). By a conversely monotonic relation \( R \) here we mean that if \( (\alpha_1,\alpha_2) \in R \) and \( \alpha_1' \leq \alpha_1 \) then there exists \( \alpha'_2 \) such that \( (\alpha_1',\alpha'_2) \in R \) and \( \alpha_2 \leq \alpha'_2 \).

One example of a construct whose input-output relation \( \text{io}(.) \) is not necessary conversely monotonic is the choice \( \sum_{i \in I} \text{when } c_i \text{ do } P_i \) when two of the guards (the \( c_i \)'s) are compatible but different. We say that two constraints \( c \) and \( d \) are compatible if there exists \( e \neq \text{false} \) such that \( e \models c \) and \( e \models d \). In literature, non-compatible constraints are also called mutually exclusive.

**Example 4.3.2 (Conversely Non-monotonic Summation).** Let us consider the following processes

\[ P = (\text{when } (x = a) \text{ do } \text{tell(true)}) + (\text{when } \text{true} \text{ do } \text{tell(y = b)}) \]

Notice that \( \text{io}(P) \) is not conversely monotonic since process \( P \) on input \( \alpha_1 = (x = a).\text{true}^\omega \) it can output \( \alpha_2 = \alpha_1 \) while on input \( \alpha'_1 = \text{true}^\omega \) it can only output \( \alpha'_2 = (y = b).\text{true}^\omega \).

By using the local operator, we can now construct a counterexample to completeness. Consider the process

\[ Q = \text{local } x \text{ in } P \]

We have \( \alpha_1 \in [P] \) and consequently, since \( \exists x \alpha'_1 = \exists x \alpha_1, \alpha'_1 \in [Q] \). However one can verify that \( \alpha'_1 \notin \text{sp}(Q) \).

\[ \Box \]
The above example corresponds to Example 4.2 in [DBGMP97], where an analogous fact is proved for ccp. In [DBGMP97] an even stronger negative result is proved: There exist no denotational semantics \( \mathcal{E} \) for ccp such that \([P]\) is equal to the strongest postcondition of \( P \) for every \( P \). Following [DBGMP97], we could prove an analogous result for ntcc.

The choice is the only ccp construct that in general does not satisfy converse monotonicity. In ntcc, however, there is also another operator which does not satisfy converse monotonicity: the \textbf{unless} construct.

\textbf{Example 4.3.3 (Conversely Non-monotonic Unless).} Let us consider the following processes

\[
\begin{align*}
P &= \text{unless } (x = a) \text{ next tell}(y = b) \\
Q &= \text{local } x \text{ in } P
\end{align*}
\]

Let \( \alpha_1 \) and \( \alpha_1' \) be like in Example 4.3.2. Note that \( P \) on input \( \alpha_1 \) gives only the output \( \text{true.(}y = b\text{)} \), \( \text{true}^2 \), and on input \( \alpha_1 \) gives only the output \( \alpha_1 \). Thus, \( \text{io}(P) \) is not conversely monotonic. Notice that \( \text{io}(P) \) is not monotonic either, and that like in Example 4.3.2, \( \alpha_1' \in [Q] \) while \( \alpha_1' \notin sp(Q) \).

\[\square\]

\subsection{4.3.1 The Locally Independent Fragment}

We consider now the conditions under which completeness does hold. Intuitively, we need to make sure that we use locality only in combination with conversely monotonic constructs. This justifies the definition of the following significant ntcc fragment.

\textbf{Definition 4.3.4 (Locally Independent Processes).} We say that \( P \) is a locally-independent (choice) process iff both the following conditions are satisfied:

1. For every construct of the form \( \sum_{i \in I} \text{when } c_i \text{ do } P_i \) occurring in \( P \), either
   \begin{itemize}
   \item \( I \) is a singleton, or
   \item the \( c_i \)'s contain only free variables, i.e. their variables are not in the scope of any local operator in \( P \).
   \end{itemize}

2. For every construct of the form \textbf{unless } \text{next } P \text{ occurring in } P, c \text{ contains only free variables}.

Another interesting fragment is the so-called restricted choice fragment considered by [FGMP97]. Restricted choice means that in every choice the guards are either pairwise mutually exclusive or equal. In fact, all the application examples in this thesis (Chapter 6) belong to either the locally-independent or the restricted choice fragment.

It turns out that for the strongest-postcondition observation, the locally-independence condition subsumes the restricted choice condition. It is not difficult to see that the strongest-postcondition of a summation with equal guards (also called internal choice) \( \sum_{i \in I} \text{when } c \text{ do } P_i \) is equivalent to that of
when \( c \) do \( \sum_{i \in I} \) when \( \text{true} \) do \( P_i \)  

As for the mutually exclusion condition, in untimed ccp, the strongest-postcondition of the (mutually exclusive) summation \( \sum_{i \in I} \) when \( c_i \) do \( P_i \) is that the same as that of

\[
\prod_{i \in I} \) when \( c_i \) do \( P_i .
\]

Nevertheless, in ntcc, the summation and the product above may not have the same strongest-postcondition. The input (or the inconsistent store) \text{false} certainly activates all the \( P_i \)'s in the product but it only activates one of them in the summation. Notice that since the \( c_i \)'s are mutually exclusive, \text{false} is the only constraint that can activate more than one of the \( P_i \)'s in the product. In the untimed case this mismatch between the product and the summation is not a problem since the output will be \text{false} anyway. In ntcc the output will also be \text{false} in the current time interval but not necessarily in the future ones. Therefore, the activity triggered by one of the \( P_i \)'s in the next time interval could be quite different from the one triggered by all the \( P_i \)'s in the next time interval. The following example should make this matter clear.

**Example 4.3.5.** Consider the following processes:

\[
\begin{align*}
P & = \sum_{i \in \{0,1\}} \) when \( x = i \) do \text{next} \text{tell}(z = i) \\
P' & = \prod_{i \in \{0,1\}} \) when \( x = i \) do \text{next} \text{tell}(z = i)
\end{align*}
\]

The reader can verify that \( \alpha = \text{false} \) \( z = 0 \), \( \alpha' \in \text{sp}(P) \) but \( \alpha \notin \text{sp}(P') \). \qed

Fortunately, we can solve the problem of the mismatch between the product and the summation on input \text{false} by using the “unless” operator. As explained below, the idea is simple: use such an operator to pre-empt any future activity that the \( P_i \)'s in the product may trigger in the store \text{false}.

Let us denote by \( \mathcal{P} \) the process that results from replacing in \( P \) each non temporal-guarded (i.e., not guarded by next, unless, replication, or the star operator) occurrence of the processes \text{next} \( Q \), \text{!} \( Q \) and \text{*} \( Q \) with the processes (unless \text{false} \text{next} \( Q \)), \( (Q \ || \) unless \text{false} \text{next} !Q) \) and (unless \text{false} \text{next} \text{*} \( Q \)) respectively. Below we define \( \mathcal{P} \) more precisely.

**Convention 4.3.6.** We shall say that a function \( h : \text{Proc} \longrightarrow \text{Proc} \) is homomorphic w.r.t the parallel operator iff \( h(P \ || \ Q) = h(P) \ || \ h(Q) \) and similarly for the other operators.

**Definition 4.3.7 (Failure Pre-emptive Encoding).** Let \( \mathcal{P} \) be the process recursively defined by

\[
\begin{align*}
\text{unless} \ c \text{next} \mathcal{P} & = \text{unless} \ c \text{next} P \\
\text{next} Q & = \text{unless} \text{false} \text{next} Q \\
\overline{Q} & = \overline{Q \ || \) unless \text{false} \text{next} !Q \\
\ast \overline{Q} & = Q + \text{unless} \text{false} \text{next} \ast Q
\end{align*}
\]

with “\( \overline{\text{—}} \)” being an homomorphism for the other operators.
Notice that in \( \mathcal{P} \) we replaced only those processes that in store false can, in the current time interval, trigger activity in the future – e.g., no unless \( c \) next \( Q \) process can trigger future activity on input false. Moreover, it is not difficult to see that each replacement does not trigger activity in the future time units if the store eventually becomes false, but it does behave the same that the original process if the store does not become false.

**Proposition 4.3.8.** For every process \( P \),

- if \( c \neq \text{false} \) then \( P \xrightarrow{(c,c)} Q \) iff \( \mathcal{P} \xrightarrow{(c,c)} Q \)
- if \( c = \text{false} \) then \( \mathcal{P} \xrightarrow{(c,c)} \text{skip} \).

**Proof.** By induction on the structure of \( P \). We confine ourself to the parallel case; the other cases can be easily verified from Definition 4.3.7. Suppose \( P = P_1 \parallel P_2 \).

Consider the first item. For the “if” direction, we have \( \mathcal{P} = \mathcal{P}_1 \parallel \mathcal{P}_2 \xrightarrow{(c,c)} Q \). The from the operational semantics of the parallel operator we can conclude that \( Q \equiv Q_1 \parallel Q_2 \) where \( Q_1 \) and \( Q_2 \) are such that \( \mathcal{P}_1 \xrightarrow{(c,c)} Q_1 \) and \( \mathcal{P}_2 \xrightarrow{(c,c)} Q_2 \). Hence, by appeal to induction we obtain (a) \( P_1 \xrightarrow{(c,c)} Q_1 \) and (b) \( P_2 \xrightarrow{(c,c)} Q_2 \). From (a) and (b) we can verify that \( P_1 \parallel P_2 \xrightarrow{(c,c)} Q_1 \parallel Q_2 \) as wanted. The “only if” can be obtaining analogously by reversing the proof of the “if” case.

As for the second item, we have \( \mathcal{P}_1 \xrightarrow{(\text{false,false})} \text{skip} \) and \( \mathcal{P}_2 \xrightarrow{(\text{false,false})} \text{skip} \) by using the induction \( \mathcal{P} = \mathcal{P}_1 \parallel \mathcal{P}_2 \). It follows from the operational semantics of the parallel operator that \( \mathcal{P}_1 \xrightarrow{(\text{false,false})} \text{skip} \) and \( \mathcal{P}_2 \xrightarrow{(\text{false,false})} \text{skip} \) then \( \mathcal{P}_1 \parallel \mathcal{P}_2 \xrightarrow{(\text{false,false})} \text{skip} \).

\( \square \)

From the above argument, one can prove that the strongest-postcondition of a mutually-exclusive summation \( \sum_{i \in I} \) when \( c_i \) do \( P_i \) is that of

\[
\prod_{i \in I} \text{when } c_i \text{ do } \mathcal{P}_i \parallel \sum_{i \in I} \text{when false do } P_i. \tag{4.10}
\]

The blind-choice summation on the right chooses one of the \( P_i \)'s if the store becomes inconsistency false. This gives us the following strongest-postcondition preserving encoding of restricted choice into the locally-independent fragment:

**Definition 4.3.9 (Encoding of Restricted-Choice).** Given a restricted-choice process \( P \), let \([P]\) be the locally-independent process defined by:

\[
\left\lfloor \sum_{i \in I} \text{when } c \text{ do } P_i \right\rfloor = \text{ when } c \text{ do } \sum_{i \in I} \text{ when true do } [P_i]
\]

\[
\left\lfloor \sum_{i \in I} \text{when } c_i \text{ do } P_i \right\rfloor = \prod_{i \in I} \text{ when } c_i \text{ do } \mathcal{P}_i \parallel \sum_{i \in I} \text{ when false do } [P_i]
\]

with \([\cdot]\) being an homomorphism for the other operators. (Above, we assume that all the \( c_i \)'s are mutually exclusive, i.e., if \( c \models c_i \) and \( c \models c_j \), \( i, j \in I, i \neq j \) then \( c = \text{false} \).)
4.3. Completeness of the Denotation

Theorem 4.3.10 (Correctness of the Encoding of Restricted-Choice). For every restricted-choice process \( P \), \( sp(P) = sp([P]) \).

Proof. Consider \( sp(P) \subseteq sp([P]) \). Let us assume that \( \alpha = c.c' \in sp(P) \). From Theorem 3.5.2, we have \( P \xrightarrow{(\alpha,\alpha)} \omega \). The proof proceeds by induction on the structure of \( P \). Here we illustrate the mutually-exclusive summation case.

Assume that \( P = \sum_{i \in I} \text{when } c_i \text{ do } P_i \) where all the \( c_i \) are mutually exclusive. In this case \( [P] = \prod_{i \in I} [P_i] \parallel \sum_{i \in I} \text{when false do } [P_i] \). From the assumption we have either (a) \( c \models c_j \) for some \( j \in I \) or (b) \( c \not\models c_i \) for all \( i \in I \). Case (b) is trivial.

Suppose (a). Let \( j \) such that \( c \models c_j \). In this case we infer that \( \langle P, c \rangle \rightarrow \langle P_j, c \rangle \) and \( \langle P_j, c \rangle \rightarrow^{*} \langle R, c \rangle \rightarrow \) for \( R \) s.t. \( F(R) \xrightarrow{(\alpha,\alpha)} \omega \). We then conclude \( P_j \xrightarrow{(\alpha,\alpha)} \omega \). By appeal to induction \( [P_j] \xrightarrow{(\alpha,\alpha)} \omega \).

From the operational semantic we obtain

\[
\langle [P], c \rangle \rightarrow \langle [P_j] \parallel Q_1 \parallel Q_2, c \rangle
\]

(4.11)

where \( Q_1 = \prod_{i \in I \setminus \{j\}} \text{when } c_i \text{ do } [P_i] \), and \( Q_2 = \sum_{i \in I} \text{when false do } [P_i] \).

Suppose \( c \not\models \text{false} \). By using the induction and Proposition 4.3.8 we obtain \( [P_j] \xrightarrow{(\alpha,\alpha)} \omega \). Since \( c \not\models \text{false} \) then \( c_j \) is the unique guard such that \( c \models c_j \).

It then follows that \( Q_1 \xrightarrow{(c,c)} \text{skip} \), and \( Q_2 \xrightarrow{(c,c)} \text{skip} \). Therefore, from Equation 4.11 we derive \( [P] \xrightarrow{(\alpha,\alpha)} \omega \).

Suppose \( c = \text{false} \). Then \( c \models c_i \) for all \( i \in I \). From Proposition 4.3.8 we derive \( [P_j] \xrightarrow{(c,c)} \text{skip} \) and similarly \( Q_1 \xrightarrow{(c,c)} \text{skip} \). From the operational semantics \( \langle Q_2, c \rangle \rightarrow \langle [P_j], c \rangle \). Thus by using the induction \( Q_2 \xrightarrow{(\alpha,\alpha)} \omega \). We then conclude \([P] \xrightarrow{(\alpha,\alpha)} \omega \) from Equation 4.11.

The case \( sp([P]) \subseteq sp(P) \) also proceeds by induction on the structure of \( P \) along the same lines of the reverse case. \( \square \)

4.3.2 The Proof of Completeness

We now turn to showing the completeness of the denotation for locally-independent processes. We need the following result showing that local independence is preserved through derivations:

Proposition 4.3.11 (Local Independence Invariance). If \( P \) is a locally-independent process, and \( P \xrightarrow{(c,d)} Q \) for some \( c, d \) and \( Q \), then also \( Q \) is locally-independent.

Proof. Reduction \( \rightarrow \), structural congruence \( \equiv \), and function \( F \) preserve local independence. \( \square \)

We also need the following technical definition and lemmas. Recall that \( C[P] \) denotes the process that results from replacing the hole in the context \( C \) with \( P \).
Definition 4.3.12 (Relation $\preceq$). Let $\preceq$ be the minimal ordering relation on processes satisfying the following rules:

\[
\begin{align*}
\preceq (\text{skip}) & \quad \text{skip} \preceq P \\
\preceq (\text{STR}) & \quad P \equiv P' \preceq Q' \equiv Q \\
\end{align*}
\]

Intuitively, $P \preceq Q$ represents the fact that $Q$ contains “at least as much code” as $P$. From a computational point of view, $Q$ is at least as active as $P$, and therefore $P$ has at least the resting points of $Q$, i.e., $sp(Q) \subseteq sp(P)$.

The following lemma shows that the “future” operator is monotonic wrt $\preceq$.

Lemma 4.3.13 ($\preceq$-Monotonicity of Future). Let $P$ and $Q$ be ntcc processes. If $P \preceq Q$, then $F(P) \preceq F(Q)$.

Proof. The proof proceeds by induction on the depth of the inference $P \preceq Q$. We consider the last step of the inference.

By using $\preceq (\text{SK})$. Then $P$ takes the form skip. Trivially $F(\text{skip}) = \text{skip}$.

By using Rule $\preceq (\text{SK})$ we obtain $F(\text{skip}) \preceq F(Q)$.

By using $\preceq (C)$. Suppose that $C$ is the context $R \parallel [\cdot]$. Therefore $P$ takes the
form $R \parallel P'$ and $Q$ takes the form $R \parallel Q'$ where $P' \preceq Q'$ by a shorter inference. By appeal to induction we obtain $F(P') \preceq F(Q')$. By definition $F(P) = F(R) || F(P')$ and $F(Q) = F(R) || F(Q')$. We can then use Rule $\preceq (C)$ to infer $F(R) || F(P') \preceq F(R) || F(Q')$. The other cases for $C$ are similar.

By using $\preceq (\text{STR})$. Then $P \equiv P'$ and $Q \equiv Q'$ where $P' \preceq Q'$ by a shorter inference. Thus, using the induction, $F(P') \preceq F(Q')$. It easy to verify that $F$ preserves $\equiv$. Hence, $F(P) = F(P')$ and $F(Q) = F(Q')$. We therefore conclude $F(P) \preceq F(Q)$.

$\square$

Lemma 4.3.14. Let $P$ be a sub-term of a local independent process $P'$, and assume that for some $c$ and some $P_1, P_2, \ldots, P_n$,

\[
\langle P_1, c \rangle \rightarrow \langle P_2, c \rangle \rightarrow \ldots \langle P_n, c \rangle \not\rightarrow .
\]

Suppose $x \notin \text{fo}(P')$. Let $Q$ and $d_1$ be s.t. $Q \preceq P$ and $c \models d_1$. Then there exist $Q_1 = Q, Q_2, \ldots, Q_m$ and $d_2, \ldots, d_m$, $m \leq n$, s.t.,:

\[
\langle Q_1, (\exists x c) \land d_1 \rangle \rightarrow \langle Q_2, (\exists x c) \land d_2 \rangle \rightarrow \ldots \langle Q_m, (\exists x c) \land d_m \rangle \not\rightarrow
\]

with $F(Q_m) \preceq F(P_n)$ and $c \models d_i$, for each $i \in \{2, \ldots, m\}$.

Proof. By induction on $n$. 

4.3. Completeness of the Denotation

$n = 1$. In this case we have $\langle P_1, c \rangle \rightarrow$. Since $c \models (\exists_x c) \land d_1$, it is easy to see, by case analysis on $P_1$, that $\langle Q_1, (\exists_x c) \land d_1 \rangle \rightarrow$. Furthermore, by Lemma 4.3.13 we get $F(P_1) \preceq F(Q_1)$.

$n > 1$. If $Q_1 = \textbf{skip}$ we are done. Otherwise we proceed by case analysis on $P_1$. We consider only the choice and “unless” operators; the other cases are easy.

$$P_1 = \sum_{j \in J} \textbf{when } e_j \textbf{ do } R_j.$$ Since $Q_1 \preceq P_1$, we therefore have $Q_1 \equiv \sum_{j \in J} \textbf{when } e_j \textbf{ do } S_j$ with $S_j \preceq R_j$ for all $j \in J$. Let $c \models e_j$, and $P_2 = R_j$. There are two cases:

$$(\exists_x c) \land d_1 \models e_j.$$ Let $Q_2 = S_j$ and $d_2 = d_1$. We therefore have $\langle Q_1, (\exists_x c) \land d_1 \rangle \rightarrow \langle Q_2, (\exists_x c) \land d_2 \rangle$. It is easy to verify that $x$ does not occur in $P_2$. Therefore, since $Q_2 \preceq P_2$ and $c \models d_2$, we can apply the inductive hypothesis to get the conclusion.

$$(\exists_x c) \land d_2 \not\models e_j.$$ Since $(\exists_x c) \not\models e_j$ and $c \models e_j$, $e_j$ must contain occurrences of $x$. Since $x$ is not a free variable, by definition of local independence $I$ must be a singleton. Thus $\langle Q_1, (\exists_x c) \land d_1 \rangle \rightarrow \langle \textbf{skip}, (\exists_x c) \land d_1 \rangle$.

Finally, note that $F(Q_1) = \textbf{skip}$.

$$P_1 = \textbf{unless } e \textbf{ next } R.$$ Since $Q_1 \preceq P_1$, $Q_1 \equiv \textbf{unless } e \textbf{ next } S$ with $S \preceq R$. If $\langle P_1, c \rangle \rightarrow \langle P'_1, c \rangle$ then $P'_1 \equiv \textbf{skip}$ and $c \models e$. Recall that $x$ does not occur free in $P'$. Thus, since $P_1$ is locally-independent, $e$ must not contain occurrences of $x$. Hence $(\exists_x c) \land d_1 \models e$ and therefore $\langle Q_1, (\exists_x c) \land d_1 \rangle \rightarrow \langle \textbf{skip}, (\exists_x c) \land d_1 \rangle$.

\qed

**Lemma 4.3.15.** Let $(\textbf{local} x) P$ be a local independent process. Assume that for some $c$ and $R$, $P \xrightarrow{(c,c)} R$. Then for every $Q$ and $d$ s.t. $Q \preceq P$ and $\exists_x d = \exists_x c$ there exists $S \preceq R$ s.t., $(\textbf{local} x) Q \xrightarrow{(d,d)} (\textbf{local} x) S$.

**Proof.** Assume $P \xrightarrow{(c,c)} R$, $Q \preceq P$, and $\exists_x d = \exists_x c$. Then there exist some processes $P_1 = P, P_2, \ldots P_n$ with $F(P_n) = R$ such that

$$\langle P_1, c \rangle \rightarrow \langle P_2, c \rangle \rightarrow \cdots \langle P_n, c \rangle \rightarrow$$

Let $d_1 = \textbf{true}$. By Lemma 4.3.14 there must be $Q_1 = Q, Q_2, \ldots Q_m$, with $m \leq n$, and constraints $d_2, \ldots d_m$ such that $c \models d_i$ for every $i$ with $2 \leq i \leq m$, and

$$\langle Q_1, (\exists_x c) \land d_1 \rangle \rightarrow \langle Q_2, (\exists_x c) \land d_2 \rangle \rightarrow \cdots \langle Q_m, (\exists_x c) \land d_m \rangle \rightarrow$$
with \( F(Q_m) \leq F(P_n) \). Since \( \exists x c = \exists x d \), by repeated application of Rule LOC, we can obtain

\[
\langle (\text{local } x, d_1) Q_1, d \land \exists x d_1 \rangle \quad \rightarrow \\
\langle (\text{local } x, d_2) Q_2, d \land \exists x d_2 \rangle \quad \rightarrow \\
\quad \ldots \\n\langle (\text{local } x, d_m) Q_m, d \land \exists x d_m \rangle \quad \not\rightarrow 
\]

Observe now that for each \( i \) such that \( 1 \leq i \leq m \) we have \( d \models \exists x d = \exists x c = \exists x d_i \) and therefore \( d \land \exists x d_i = d \). Hence we obtain

\[
(\text{local } x) Q_1 \xrightarrow{(d,d)} F((\text{local } x, d_m) Q_m) = (\text{local } x) F(Q_m)
\]

Finally, define \( S = F(Q_m) \) and note that \( S = F(Q_m) \leq F(P_n) = R \).  

We are now ready to prove our main result:

**Theorem 4.3.16 (Completeness).** For every locally-independent process \( P \), we have \( \llbracket P \rrbracket \subseteq \text{sp}(P) \).

**Proof.** Let us assume that \( \alpha \in \llbracket P \rrbracket \) where \( P \) is locally independent. We shall show that \( P \xrightarrow{(a,\alpha)} \), i.e., \( \alpha \in \text{sp}(P) \) by Theorem 3.5.2. The proof proceeds by induction on the structure of \( P \). The only case for which we need the local independence condition is the \text{local} operator, and we will discuss this case thoroughly. The cases for the operators \text{next}, \|, \text{!} and \* can be proved easily by reversing the proofs of the corresponding cases in Theorem 4.2.1 and so we skip them. As for the remaining cases, we give their proofs in full extension, although they are also similar to the reverse of the proofs in Theorem 4.2.1 except for some technical details.

\( P = \text{local } x \text{ in } P' \). Assume \( \alpha \in \llbracket \text{local } x \text{ in } P' \rrbracket \) and let \( \alpha = c_1, c_2, \ldots, c_n, \ldots \) . By definition of \( \llbracket \text{local } x \text{ in } P' \rrbracket \), there exist \( \alpha' = c'_1, c'_2, \ldots, c'_n, \ldots \) such that \( \exists x c'_i = \exists x c_i \) for every \( i \geq 1 \) and \( \alpha' \in \llbracket P' \rrbracket \). By inductive hypothesis, we must have a derivation of the form

\[
P' = P'_1 \xrightarrow{(c'_1, c'_1)} P'_2 \xrightarrow{(c'_2, c'_2)} \ldots P'_n \xrightarrow{(c'_n, c'_n)} \ldots
\]

By Proposition 4.3.11, each \( P'_i \) is locally independent. Hence, by repeated application of Lemma 4.3.15, we derive that there exist \( Q_1, Q_2, \ldots, Q_n, \ldots \) with \( Q_1 = P' \) such that \( Q_i \preceq P'_i \) for every \( i \geq 2 \) and

\[
\text{local } x \text{ in } P' = \underbrace{\text{local } x \text{ in } Q_1 \xrightarrow{(c_1, c_1)}}_{\substack{\text{local } x \text{ in } Q_2 \xrightarrow{(c_2, c_2)} \quad \ldots \quad \text{local } x \text{ in } Q_n \xrightarrow{(c_n, c_n)} \quad \ldots}}_{\text{local } x \text{ in } P'} \xrightarrow{(a, \alpha)} \omega.
\]

Hence we conclude \( P = \text{local } x \text{ in } P' \xrightarrow{(a, \alpha)} \omega \).
Let $P = \text{tell}(c)$. Assume $\alpha \in \llbracket \text{tell}(c) \rrbracket$. Then $\alpha = d.\alpha'$ with $d \models c$. Hence, we have $\text{tell}(c) \xrightarrow{(d,d)} \text{skip}$. Since $\text{skip} \xrightarrow{(\alpha',\alpha')} \omega$, we can conclude that $\text{tell}(c) \xrightarrow{(d,d),\alpha'} \omega$.

$P = \sum_{i \in I} \text{when } c_i \text{ do } P_i$. Assume $\alpha \in \llbracket \sum_{i \in I} \text{when } c_i \text{ do } P_i \rrbracket$. Let $\alpha = d.\alpha'$. We distinguish two cases:

1. There exists $i \in I$ such that $d \models c_i$ and $d.\alpha' \in \llbracket P_i \rrbracket$. In this case, we have $(P,d) \rightarrow (P_i,d)$ and, by inductive hypothesis, $P_i \xrightarrow{(d,\alpha',d.\alpha')} \omega$. Hence we conclude $P \xrightarrow{(d,\alpha',d.\alpha')} \omega$.

2. For all $i \in I$, $d \not\models c_i$. Then $(P,d) \rightarrow$. Since $F(P) = \text{skip}$, we derive

$$P \xrightarrow{(d,d)} \text{skip} \xrightarrow{(\alpha',\alpha')} \omega$$

and therefore $P \xrightarrow{(d,\alpha',d.\alpha')} \omega$.

$P = \text{unless } c \text{ next } P'$. Assume $\alpha \in \llbracket \text{unless } c \text{ next } P' \rrbracket$ and $\alpha = d.\alpha'$. We distinguish two cases:

1. $d \models c$. Then $(\text{unless } c \text{ next } P',d) \rightarrow (\text{skip},d)$. Since $\text{skip} \xrightarrow{(d,\alpha',d.\alpha')} \omega$, we conclude that $\text{unless } c \text{ next } P' \xrightarrow{(d,\alpha',d.\alpha')} \omega$.

2. $d \not\models c$ and $\alpha' \in \llbracket P' \rrbracket$. In this case we should certainly have $\text{unless } c \text{ next } P' \xrightarrow{(d,d)} P'$ and, by using the inductive hypothesis, $P' \xrightarrow{(\alpha',\alpha')} \omega$. We then conclude $\text{unless } c \text{ next } P' \xrightarrow{(d,\alpha',d.\alpha')} \omega$.

$\square$

From the soundness and completeness results it follows that $P \sim sp Q$ iff $[P] = [Q]$ if $P$ and $Q$ are locally independent. Furthermore, our denotational semantics is compositional. Hence, if we confine ourselves to locally independent processes, then $\sim sp$ (Definition 3.5.4) is a congruence as stated in following full-abstraction result.

**Corollary 4.3.17 (Full-Abstraction for Local Independence).** $[P] = [Q]$ iff for all context $C[.]$, s.t. $C[P]$ and $C[Q]$ are locally-independent, we have $C[P] \sim sp C[Q]$.

### 4.4 Determinism and Monotonicity

In this section we investigate the semantic properties of another special class of ntcc processes: deterministic and monotonic processes.
Definition 4.4.1 (Deterministic & Monotonic Processes). Let $P$ be a ntcc process,

- $P$ is deterministic if it does not contain occurrences of either the choice operator (except when the index set is a singleton) or the $*$ operator.
- $P$ is monotonic if it does not contain the unless operator.

We will show that for processes which are both deterministic and monotonic the semantics allows us to retrieve the input-output relation (which for deterministic processes is a function). We first need to introduce the following definitions.

Notation 4.4.2 (Lattice Notations).

- Given a set $S \subseteq \mathcal{C}^\omega$, $\min(S)$ denotes the minimal element of $S$, if it exists.
- Given $\alpha \in \mathcal{C}^\omega$, $\uparrow \alpha$ denotes the upward closure of $\alpha$, namely
  \[ \uparrow \alpha = \{ \alpha' \mid \alpha \leq \alpha' \} \]

It is a routine exercise to show that if $(\mathcal{C}, \models)$ is a complete lattice, then also $(\mathcal{C}^\omega, \leq)$ is a complete lattice.

We can prove for deterministic and monotonic processes a property analogous to what holds for deterministic ccp, namely that the input-output relation is a closure operator on $\leq$. More precisely:

Proposition 4.4.3 (Deterministic & Monotonic Properties). If $P$ is a deterministic and monotonic process, then

1. $\text{io}(P)$ is a function, i.e., if $(\alpha, \alpha') \in \text{io}(P)$ and $(\alpha, \alpha'') \in \text{io}(P)$ then $\alpha' = \alpha''$.

2. $\text{io}(P)$ is a closure operator, namely it satisfies the following properties

   - Extensiveness: If $(\alpha, \alpha') \in \text{io}(P)$ then $\alpha \leq \alpha'$.
   - Idempotency: If $(\alpha, \alpha') \in \text{io}(P)$ then $(\alpha', \alpha') \in \text{io}(P)$.
   - Monotonicity: If $(\alpha_1, \alpha_2) \in \text{io}(P)$ and $\alpha_1 \leq \alpha_1'$, then there exists $\alpha_2'$ such that $(\alpha_1', \alpha_2') \in \text{io}(P)$ and $\alpha_2 \leq \alpha_2'$.

Proof. 1. Analogous to the standard case for deterministic ccp (see for instance [SRP91]). Note that the interleaving rule for the parallel operator does not introduce any nondeterminism.

2. Extensiveness: From Theorem 3.4.8 – this property holds for ntcc processes in general.

   Idempotency: From Theorem 3.4.8 – also holds for ntcc processes in general.
**Monotonicity:** It is sufficient to show that for every $Q$, $R$, $c_1$ and $c_2$, if $\langle Q, c_1 \rangle \longrightarrow^* \langle R, c_2 \rangle$, then for every $c'_1 \models c_1$ and $Q'$ such that $Q \preceq Q'$ there exists $c'_2 \models c_2$ and $R'$ with $F(R) \preceq F(R')$ such that $\langle Q', c'_1 \rangle \longrightarrow^* \langle R', c'_2 \rangle$. This can be proved by induction on the length of the derivation using the following two properties:

(a) $\longrightarrow$ is monotonic wrt the store, in the sense that, for every $Q$, $R$, $c_1$ and $c_2$, if $\langle Q, c_1 \rangle \longrightarrow \langle R, c_2 \rangle$ then for every $c'_1 \models c_1$ and $Q'$ such that $Q \preceq Q'$ there exists $c'_2 \models c_2$ and $R'$ with $R \preceq R'$ such that $\langle Q', c'_1 \rangle \longrightarrow \langle R', c'_2 \rangle$. This property holds for ntcc in general and can be proved easily by induction on the structure of $Q'$.

(b) For every monotonic $Q$ and $c_1$, if $\langle Q, c_1 \rangle \not\longrightarrow$ then for every $c'_1 \models c_1$ and $Q'$ such that $Q \preceq Q'$ we have either

- $\langle Q', c'_1 \rangle \not\longrightarrow$, or
- There exists $c'_2 \models c_1$ and $R'$ with $F(Q) \preceq F(R')$ such that $\langle Q', c'_1 \rangle \longrightarrow^* \langle R', c'_2 \rangle$. Also this property can be proved easily by induction on the structure of $Q'$. The restriction to programs which do not contain **unless** constructs is essential here.

Note that **unless** is the only ntcc operator which introduces non-monotonicity across time boundaries. All the other ntcc constructs, including those of ccp, are monotonic.

A pleasant property of closure operators over a complete lattice is that they can be completely characterized by the set of their fixed points (see [SRP91] for details). In our case these fixed-points are the elements of $sp(P)$. This characterization is expressed by the following lemma, whose proof is standard, given that $io(P)$ is a closure operator.

**Lemma 4.4.4.** If $P$ is a deterministic and monotonic process, then $(\alpha, \alpha') \in io(P)$ iff $\alpha' = min(sp(P) \cap \uparrow \alpha)$.

**Proof.** Follows from Proposition 4.4.3 and standard results from closure operators (see [SRP91]).

With the help of the above corollary, we derive the following theorem.

**Theorem 4.4.5 (IO Retrieval from the Denotation).** If $P$ is deterministic and monotonic, then $(\alpha, \alpha') \in io(P)$ iff $\alpha' = min([P] \cap \uparrow \alpha)$.

**Proof.** It follows from the soundness and completeness results of the denotation (Theorems 4.2.1 and 4.3.16) and Lemma 4.4.4 above.

From the above theorem we can conclude the input-output and strongest postcondition observations of deterministic and monotonic processes coincide.

**Corollary 4.4.6 (IO-SP Correspondence).** Let $P$ and $Q$ be deterministic and monotonic processes. $P \sim_{io} Q$ iff $P \sim_{sp} Q$.
4.5 Summary and Related Work

In this chapter we gave a denotational semantics for approximating the strongest-postcondition of ntcc processes. We proved the soundness of the denotation. We also identified a substantial fragment of the calculus for which completeness holds. This fragment is given by those processes which we called locally independent. These are processes in which the guards of the choice and the unless operators do not depend on local variables. We also concluded that the strongest-postcondition equivalence is a congruence for locally independent processes.

Our denotational semantics follows ideas of [dBPP95b] for untimed ccp which also gives a completeness result for the restricted choice. In such a fragment only either blind or mutually exclusive choice is allowed. We proved that restricted choice fragment for ntcc is subsumed by the local independent fragment wrt to the strongest-postcondition observation. Our completeness result can be easily adapted to the untimed case.

We also explored some relevant semantic properties of deterministic and monotonic processes by following similar results in [dBPP95b] and [SJG94a] for deterministic untimed ccp and tcc, respectively. These properties allowed us to prove that the input-output and the strongest-postcondition observation coincide for such processes.

The denotation of ntcc presented in this chapter was originally published as [PV01]. In addition to the work in [PV01], in this chapter we proved that the locally independent fragment subsumes the restricted-choice fragment.
Chapter 5

Logic and Specification

Logic takes care of itself; all we have to do is to look and see how it does it.
— Ludwig Wittgenstein

We have mentioned that \texttt{ntcc} processes can be used to specify properties of timed systems, e.g., that an action must happen within some finite but not fixed amount of time. It is often convenient, however, to express specifications in another formalism, in particular a logical one. In this chapter we shall address this issue.

We shall start by defining a linear-time temporal logic to expresses temporal properties over infinite sequences of constraints. We then define what it means for a process to satisfy a specification given as a formula in this logic. We do this by adapting to our case Dijkstra's strongest postcondition approach. We shall then say that \( P \) satisfies a specification \( A \) iff every infinite sequence \( P \) can possibly output (on inputs from arbitrary environments) satisfies \( A \), i.e., iff the strongest-postcondition of \( P \) implies \( A \).

Furthermore, we also give an inference system aimed at proving whether a process satisfies a given specification. We shall appeal to the denotational semantics given in the previous section for proving soundness and completeness results for this system.

5.1 A Linear-Temporal Logic

The syntax of our linear-temporal logic is given by the following definition.

**Definition 5.1.1 (Logic Syntax).** The formulae \( A, B, \ldots \in A \) are defined by the grammar

\[
A, B, \ldots ::= c \mid A \Rightarrow A \mid \neg A \mid \exists x. A \mid \diamond A \mid \Box A
\]

Here \( c \) denotes an arbitrary constraint which we shall refer to as atomic proposition. The intended meaning of the other symbols is the following: \( \Rightarrow \), \( \neg \) and \( \exists \) represent linear-temporal logic implication, negation and existential quantification. These symbols are not to be confused with the symbols \( \Rightarrow, \neg \) and \( \exists \) of the underlying constraint system. The symbols \( \diamond, \Box, \) and \( \Diamond \) denote the temporal operators next, always and sometime. Intuitively \( \diamond A, \Diamond A \) and \( \Box A \) means that the property \( A \) must hold next, eventually and always, respectively.
Notation 5.1.2. We use $A \lor B$ as an abbreviation of $\neg A \Rightarrow B$ and $A \land B$ as an abbreviation of $\neg (\neg A \lor \neg B)$. The temporal true symbol $\textsf{true}$ stands for $\Box \textsf{true}$ and the temporal false symbol $\textsf{false}$ stands for $\neg \textsf{true}$.

The standard interpretation structures of linear temporal logic are infinite sequences of states [MP91]. In the case of $\textsf{ntcc}$, it is natural to replace states by constraints, and consider therefore as interpretations the elements of $C^\omega$.

The semantics of the logic is given in Definition 5.1.4. Following [MP91] we first introduce the notion of $x$-variant. Recall that given a sequence $\alpha \in C^\omega$, $\exists_x \alpha$ denotes the sequence resulting from the application of $\exists_x$ to each constraint in $\alpha$. Also recall that $\alpha(i)$ denotes the $i$-th elements of $\alpha$.

Definition 5.1.3 ($x$-variants). Given $c, d \in C$ we say that $d$ is an $x$-variant of $c$ iff $\exists_x c = \exists_x d$. Similarly, given $\alpha, \alpha' \in C^\omega$ we say that $\alpha'$ is an $x$-variant of $\alpha$ iff $\exists_x \alpha = \exists_x \alpha'$

Intuitively, $d$ and $\alpha'$ are $x$-variants of $\alpha$ and $c$, respectively, if they are the same except for the information about $x$. For example, $y = 0$ is and $x$-variant of $x = y = 0$.

Definition 5.1.4 (Temporal Semantics). We say that $\alpha \in C^\omega$ is a model of (or that it satisfies) $A$, notation $\alpha \models A$, if $\langle \alpha, 1 \rangle \models A$, where:

\[
\begin{align*}
\langle \alpha, i \rangle & \models c \quad \text{iff} \quad \alpha(i) \models c \\
\langle \alpha, i \rangle & \models \neg A \quad \text{iff} \quad (\alpha, i) \not\models A \\
\langle \alpha, i \rangle & \models A_1 \Rightarrow A_2 \quad \text{iff} \quad (\alpha, i) \models A_1 \text{ implies } (\alpha, i) \models A_2 \\
\langle \alpha, i \rangle & \models \Box A \quad \text{iff} \quad (\alpha, i + 1) \models A \\
\langle \alpha, i \rangle & \models \Diamond A \quad \text{iff} \quad \text{there is a } j \geq i \text{ s.t. } (\alpha, j) \models A \\
\langle \alpha, i \rangle & \models \exists_x A \quad \text{iff} \quad \text{there is an } x \text{-variant } \alpha' \text{ of } \alpha \text{ s.t. } (\alpha', i) \models A.
\end{align*}
\]

We define $[A]$ to be the collection of all models of $A$, i.e., $[A] = \{ \alpha \mid \alpha \models A \}$.

We ought to clarify the role of constraints as atomic propositions in our logic.

A temporal formula $A$ expresses properties over sequences of constraints. As an atomic proposition, $c$ expresses a property which is satisfied only by those $e, \alpha'$ such that $e \models c$ holds. Therefore, the atomic proposition $\textsf{false}$ (and consequently $\Box \textsf{false}$) has at least one sequence that satisfies it (e.g. $\textsf{false}^\omega$). On the contrary the temporal formula $\textsf{false}$ has no models whatsoever.

Similarly, the models of the temporal formula $c \lor d$ are those $e, \alpha'$ such that either $e \models c$ or $e \models d$ holds. Therefore, the formula $c \lor d$ and the atomic proposition $c \lor d$ may have different models since, in general, one can verify that $e \models c \lor d$ may hold while neither $e \models c$ nor $e \models d$ hold – e.g. take $e = (x = 1 \lor x = 2)$, $c = (x = 1)$ and $d = (x = 2)$.

In contrast, the formula $c \land d$ and the atomic proposition $c \land d$ have the same models since $e \models (c \land d)$ holds if and only if both $e \models c$ and $e \models d$ hold.

The above discussion tells us that the operators of the constraint system should not be confused with those of the temporal logic. In particular, the
operators \( \lor \) and \( \dot{\lor} \). This distinction does not make our logic intuitionistic. In fact, classically but not intuitionistically valid statements such as \( \neg A \dot{\lor} A \) and \( \dot{\lor} \neg A \Rightarrow A \) are also valid in our logic (i.e., all sequences in \( C^\omega \) are models of these statements).

5.2 Specification

In this section we use our logic as a formal specification language. Let us start by defining what it means for a process \( P \) to satisfy a specification \( A \).

Definition 5.2.1 \((P \models A)\). Given a ncc process \( P \), and a temporal logic formula \( A \), we say that \( P \) satisfies \( A \), notation \( P \models A \), iff \( sp(P) \subseteq [A] \).

Thus, the intended meaning of \( P \models A \) is that every sequence \( P \) can possibly output on inputs from arbitrary environments satisfies the temporal formula \( A \). For example, \( \# \text{tell}(c) \models \Diamond c \) since in every infinite sequence output by \( \# \text{tell}(c) \) on arbitrary inputs there must be an element entailing \( c \).

Notice that \( P = \text{tell}(c) + \text{tell}(d) \models (c \lor d) \) as every constraint \( e \) output by \( P \) entails either \( c \) or \( d \). In contrast, \( Q = \text{tell}(c \land d) \not\models (c \lor d) \) in general since \( Q \) can output a constraint \( e \) which certainly entails \( c \land d \) and still entails neither \( c \) nor \( d \) — e.g., consider \( e = (x = 1 \lor x = 2) \), \( c = (x = 1) \) and \( d = (x = 2) \). Notice, however, that \( Q \models (c \lor d) \). In other words the formula \( c \lor d \) distinguishes \( P \) from \( Q \).

The reader may now see why we wish to distinguish the temporal formula \( c \lor d \) from the atomic proposition \( c \land d \).

5.2.1 Proof System

In order to reason about statements of the form \( P \models A \), we propose a proof (or inference) system for assertions of the form \( P \vdash A \). Intuitively, we want \( P \vdash A \) to be the “counterpart” of \( P \models A \) in the inference system, namely \( P \vdash A \) should approximate \( P \models A \) as closely as possible (ideally, they should be equivalent). The system is presented in Table 5.1.

Definition 5.2.2 \((P \vdash A)\). We say that \( P \vdash A \) iff the assertion \( P \vdash A \) has a proof in the system in Table 5.1.

Inference Rules

Let us now describe some of the inference rules of the proof system. The inference rule for the tell operator is given by

\[
\text{LETLL} : \quad \text{tell}(c) \vdash c
\] (5.1)

Rule LETLL gives a proof saying that every output of \( \text{tell}(c) \) on inputs of arbitrary environments should definitely satisfy the atomic proposition \( c \), i.e., \( \text{tell}(c) \models c \).
Consider now the rule for the choice operator:

\[
\text{LSUM:} \quad \forall i \in I \quad P_i \vdash A_i
\]
\[
\sum_{i \in I} \text{when } c_i \text{ do } P_i \vdash \bigvee_{i \in I} (c_i \land A_i) \lor \bigvee_{i \in I} \neg c_i
\]  

Rule LSUM can be explained as follows. Suppose that given a summation \( P = \sum_{i \in I} \text{when } c_i \text{ do } P_i \) we are given a proof that each \( P_i \) satisfies \( A_i \). Then we certainly have a proof saying that every output of \( P \) on arbitrary inputs should satisfy either: (a) some of the guards \( c_i \) and their corresponding \( A_i \) (i.e., \( \bigvee_{i \in I} (c_i \land A_i) \)), or (b) none of the guards (i.e., \( \bigwedge_{i \in I} \neg c_i \)).

The inference rule for parallel composition is defined as

\[
\text{LPAR:} \quad \begin{array}{c} P \vdash A \\ Q \vdash B \end{array} \quad \frac{}{P \parallel Q \vdash A \land B}
\]

The soundness of this rule can be justified as follows. Assume that each output of \( P \), under the influence of arbitrary environments, satisfies \( A \). Assume the same about \( Q \) and \( B \). In \( P \parallel Q \), the process \( Q \) can be thought as one of those arbitrary environment under which \( P \) satisfies \( A \). Then \( P \parallel Q \) must satisfy \( A \). Similarly, \( P \) can be one of those arbitrary environment under which \( Q \) satisfies \( B \). Hence, \( P \parallel Q \) must satisfy \( B \) as well. We therefore have grounds to conclude that \( P \parallel Q \) satisfies \( A \land B \).

The inference rule for the local operator is

\[
\text{LLOC:} \quad \begin{array}{c} P \vdash A \end{array} \quad \frac{}{(\text{local} \ x) \ P \vdash \exists x A}
\]

The intuition is that since the outputs of \((\text{local} \ x) \ P\) are outputs of \( P \) with \( x \) hidden then if \( P \) satisfies \( A \), \((\text{local} \ x) \ P\) should satisfy \( A \) with \( x \) hidden, i.e., \( \exists x A \).

The following are the inference rules for the temporal \texttt{ntcc} constructs:

\[
\text{LNEXT:} \quad \begin{array}{c} P \vdash A \end{array} \quad \frac{}{\text{next } P \vdash \diamond A}
\]
\[
\text{LUNL:} \quad \begin{array}{c} P \vdash A \end{array} \quad \frac{}{\text{unless } c \text{ next } P \vdash c \lor A}
\]
\[
\text{LREP:} \quad \begin{array}{c} P \vdash A \end{array} \quad \frac{}{P \vdash A}
\]
\[
\text{LSTAR:} \quad \begin{array}{c} P \vdash A \end{array} \quad \frac{}{P \vdash A}
\]

Assume that \( P \) satisfies \( A \). Rule LNEXT says that if \( P \) is executed next, then in the next time unit it will also satisfy \( A \). Hence, \( \text{next } P \) satisfies \( \diamond A \). Rule LUNL is similar, except that \( P \) can also be precluded from execution if some environment provides \( c \). Thus \( \text{unless } c \text{ next } P \) satisfies either \( c \) or \( \diamond A \). Rule
LREP says that if $P$ is executed in each time interval, then $A$ is always satisfied by $P$. Therefore, $!P$ satisfies $\Box A$. Rule LSTAR says that if $P$ is executed in some time interval, then in that time interval $P$ satisfies $A$. Therefore, $\star P$ satisfies $\diamond A$.

Finally, we have a rule that allows reasoning about temporal formulae to be incorporated in proofs about processes satisfying specifications:

\[
\text{LCONS: } \frac{P \models A}{P \models B} \quad \text{if } A \Rightarrow B \tag{5.9}
\]

Rule LCONS simply says that if $P$ satisfies a specification $A$ then it also satisfies any weaker specification $B$. We shall also refer to LCONS as the consequence rule.

Notice that the inference rules reveal a pleasant correspondence between ntcc operators and the logic operators. For example, parallel composition and locality corresponds to conjunction and existential quantification. The choice operator corresponds to some special kind of conjunction. The next, replication and star operators correspond to the next, always, and eventuality temporal operator.

### 5.2.2 Derivation Example

We shall now give a simple example illustrating a proof in the inference system.

**Example 5.2.3.** Recall Example 2.2.8. We had a process $R$ which was repeatedly checking the state of motor$_1$. If a malfunction is reported, $R$ would tell that motor$_1$ must be turned off. We also had a process $S$ stating that motor motor$_1$ was doomed to malfunction. Processes $R$ and $S$ were defined as:

\[
\begin{align*}
R &= !\text{when } c \text{ do tell}(e) \\
S &= \star \text{tell}(c)
\end{align*}
\]

where $c = \text{malfunction(motor$_1$.status)}$ and $e = (\text{motor$_1$.speed} = 0)$. We want to provide a proof of the assertion:

\[
R \parallel S \vdash \diamond e.
\]

Intuitively, this means that the parallel execution of $R$ and $S$ satisfies the specification stating that motor$_1$ is eventually turned off. The following is a derivation of the above assertion.

\[
\begin{align*}
\text{when } c \text{ do tell}(e) & \vdash (c \land e) \lor \neg c & \text{LSUM} \\
\text{when } c \text{ do tell}(e) & \vdash c \Rightarrow e & \text{LCONS} \\
R & \vdash \Box (c \Rightarrow e) & \text{LREP} \\
S & \vdash \diamond c & \text{LSTAR} \\
\hline
R \parallel S & \vdash \Box (c \Rightarrow e) \land \diamond c & \text{LPAR} \\
\hline
\end{align*}
\]

A more complex example of the use of the proof system for proving the satisfaction of processes specification will be given in Section 6.2.
Table 5.1: A proof system for linear-temporal properties of ntcc processes

5.2.3 Soundness and Completeness of the Proof System

This section is devoted to exploring the soundness and completeness of the inference system. We shall also show some other relevant properties of the proof system.

We begin by introducing the concept of *strongest temporal formula derivable for a process.*

**Definition 5.2.4 (Strongest Temporal Formula).** A formula $A$ is the strongest temporal formula derivable for a process $P$ if

1. $P \vdash A$ and
2. for every formula $A'$ such that $P \vdash A'$, we have $A \Rightarrow A'$.

Note that the strongest temporal formula of a process $P$, if it exists, is unique modulo logical equivalence. We give now a constructive proof of the existence of such a formula for every process $P$.
Definition 5.2.5 ($stf(P)$). The function $stf : \text{Proc} \rightarrow A$ is defined as follows:

\[
\begin{align*}
stf(\text{tell}(c)) &= c \\
stf(\sum_{i \in I} \text{when} \ (c_i) \text{ do } P_i) &= \left( \bigvee_{i \in I} c_i \land stf(P_i) \right) \lor \bigwedge_{i \in I} \neg c_i \\
stf(P \parallel Q) &= stf(P) \land stf(Q) \\
stf(\text{local } x \text{ in } P) &= \exists_x stf(P) \\
stf(\text{next } P) &= \circ stf(P) \\
stf(\text{unless } c \text{ next } P) &= c \lor \circ stf(P) \\
stf(\text{! } P) &= \Box stf(P) \\
stf(\star P) &= \Diamond stf(P)
\end{align*}
\]

The following result states that $stf(P)$ is the formula characterized by Definition 5.2.4.

Theorem 5.2.6. For every process $P$, $stf(P)$ is the strongest temporal formula derivable for $P$.

Proof. The proof of $P \vdash stf(P)$ proceeds by structural induction on $P$. The proof that $P \vdash A$ implies $stf(P) \Rightarrow A$ proceeds by induction on (the depth) of the inference; considering all possible cases for the final step of the inference of $P \vdash A$. Here we consider the case in which the final step is given by LPAR.

Thus $P$ takes the form $P_1 \parallel P_2$ and $A$ takes the form $A_1 \land A_2$ with $P_1 \vdash A_1$ and $P_2 \vdash A_2$ by a shorter inference. We can then use induction to conclude that $stf(P_1) \Rightarrow A_1$ and $stf(P_2) \Rightarrow A_2$. Furthermore, from Definition 5.2.5 we have $stf(P) = stf(P_1) \land stf(P_2)$. So, $stf(P) \Rightarrow A_1$ and $stf(P) \Rightarrow A_2$, hence $stf(P) \Rightarrow (A_1 \land A_2)$.

The following Corollary gives us an alternative way to determine whether $P \vdash A$ holds; determining whether $stf(P) \Rightarrow A$ is valid.

Corollary 5.2.7. $P \vdash A$ if and only if $stf(P) \Rightarrow A$.

Proof. The “only if” direction follows from the above theorem. The “if” direction follows from the rule LCONS.

The proposition below shows a complete correspondence between the denotational semantics of $P$ and the models of $stf(P)$.

Proposition 5.2.8. For every process $P$, $[P] = [stf(P)]$.

Proof. By structural induction on $P$.

The following lemma establishes that in the inference system every processes can be proven to satisfy the “always true” temporal formula, and no process can be proven to satisfy the “always false” temporal formula. It also gives us some derived inference rules which will come in handy when proving properties about processes.

Lemma 5.2.9. For every process $P$,
1. \( P \vdash \text{true} \)

2. \( P \not\vdash \text{false} \)

3. \[
\begin{align*}
P \vdash A \\
\frac{P \parallel Q \vdash A}{P \parallel Q \vdash A} \text{ LPAR}
\end{align*}
\]

4. \[
\begin{align*}
P \vdash A \\
\frac{P \parallel Q \vdash A}{P \vdash A \land B} \text{ LCONS}
\end{align*}
\]

Proof. (1) Trivially, \( \text{stf}(A) \Rightarrow \text{true} \). Thus, from Corollary 5.2.7, \( P \vdash \text{true} \).

(2) Suppose that for some \( P \), \( P \vdash \text{false} \). We have \( [\text{false}] = \emptyset \). From the operational semantics it is easy to see that every process \( P \), on any input it produces some output. Hence \( [\text{false}] \subseteq \text{sp}(P) \). From the soundness of the denotational semantics (Theorem 4.2.1) and Proposition 5.2.8 we have \( \text{sp}(P) \subseteq [P] = [\text{stf}(P)] \). From Theorem 5.2.6 \( \text{stf}(P) \) is the strongest formula derivable for \( P \), thus by definition of \( \Rightarrow \), \( [\text{stf}(P)] \subseteq [\text{false}] \), a contradiction.

(3) We have the following derivation

\[
\begin{align*}
P \vdash A \quad Q \vdash \text{true} \\
\frac{P \parallel Q \vdash A \land \text{true}}{P \parallel Q \vdash A} \text{ LCONS}
\end{align*}
\]

(4) From Corollary 5.2.7 it follows that \( \text{stf}(P) \Rightarrow A \land B \). Using Corollary 5.2.7 we obtain \( P \vdash A \land B \).

Finally, we have the following result, which states the correspondence between \( P \vdash A \) and the semantics counterparts of \( P \) and \( A \):

**Theorem 5.2.10.** For every ntcc process \( P \) and every formula \( A \), \( P \vdash A \) iff \( [P] \subseteq [A] \).

Proof.

\[
\begin{align*}
P \vdash A \\
\text{iff} \quad \text{(by Corollary 5.2.7)} \\
\text{stf}(P) \Rightarrow A \\
\text{iff} \quad \text{(by definition of \( \Rightarrow \))} \\
[P] \subseteq [A] \\
\text{iff} \quad \text{(by Proposition 5.2.8)} \\
\text{iff} \quad \text{(by Corollary 5.2.7)}
\end{align*}
\]

From Theorems 5.2.10, 4.2.1 and 4.3.16 we immediately derive the following relations between \( |\vdash | \) and \( \vdash \):

**Corollary 5.2.11 (Soundness of the Proof System).**

For every ntcc process \( P \) and every formula \( A \), if \( P \vdash A \) then \( P \vdash A \).

**Corollary 5.2.12 (Relative Completeness of the Proof System).** For every local independent process \( P \) and every formula \( A \), if \( P \vdash A \) then \( P \vdash A \).
5.3 Summary and Related Work

In this chapter we introduced a logic for expressing process specifications. We also provided a proof system for assertions about processes satisfying specifications given in this logic. Such a proof system was proven to be relative complete for locally independent processes.

Our work in this chapter extends and strengthens the work of [FGMP97]. In such work neither infinite nor timed computations are considered. Also, the completeness result of [FGMP97] is given only for the (untimed) restricted choice fragment which is subsumed by the locally independent fragment. Our completeness result can be easily adapted to the untimed case to obtain a stronger result.

In [SJG94a] the authors proposed a proof system for tcc, based on an intuitionistic logic enriched with a next operator. The system, however, is complete only for hiding-free and finite processes. In contrast our system is based on the standard classical temporal logic of [MP91] and the completeness result does not rule out hiding or infinite processes.

A proof system for tcep, which is another timed ccp formalism (see Section 9 for a brief description of it), was recently introduced in [dBGC01]. The underlying second-order linear temporal logic in [dBGC01] can be used for describing input-output behavior. In contrast, our logic can only be used for the strongest-postcondition, but also it is semantically simpler and defined as the standard first-order linear-temporal logic of [MP91]. This may come in handy when using the Consequence Rule which is also present in [dBGC01].

The material of this chapter was originally published as [PV01].
Chapter 6

Applications

To persuade me of the merit of your language, you must show me how to construct programs in it.
— Robert W. Floyd

In this chapter we illustrate some application examples of ntcunarachna based on the theory we have seen in the previous chapters. We start by encoding some basic form of recursion which will allows us to write our applications in a more succinct and clear way. We then model persistent and mutable data structures such a cells. We also model the behavior of robotic devices (the Zigzagging Robot), multi-agent systems (the Predator/Prey game) and some musical applications (generation of rhythms patterns and controlled improvisation). We shall use the logic and inference system for specifying and proving properties of these examples.

As in the previous examples, here we assume that $A[n]$ (Definition 3.1.5) is the underlying constraint system. Recall that the intended meaning of $A[n]$ is the natural numbers interpreted as in arithmetic modulo $n$. We designate $D$ as the set $\{0, 1, ..., n - 1\}$ and use $v$ and $w$ to range over its elements. For simplicity, we shall often just write $v$ instead of the expression $v \in D$.

6.1 Recursion.

Often it is convenient to specify behavior by using recursive definitions. In this section we illustrate how to encode in ntcunarachna a basic form of recursion. Namely, we consider recursive definitions of the form

$$q(x) \overset{\text{def}}{=} P_q$$  \hspace{1cm} (6.1)

where $q$ is the process name and $P_q$ is restricted to call $q$ at most once and such a call must be within the scope of a “next”. The reason for such a restriction is that we want to keep bounded the response time of the system: we do not want $P_q$ to make infinitely or unboundedly many recursive calls of $q$ within the same time interval. Furthermore, for our purposes, we need to consider call-by-value in the sense that the intended behavior of a call $q(t)$, where $t$ is a term fixed to a value $v$ (i.e., the current store entails $t = v$), is that of $P_q[v/x]$, where $[v/x]$ is the operation of (syntactical) replacement of every occurrence of $x$ by $v$.  

81
As in the $\pi$-calculus we do not want, when unfolding recursive calls of $q$, the free variables of $P_q$ to get captured in the lexical-scope of a bound-variable in $P_q$. In other words, we want static scoping rather than dynamic scoping. The following example should make this matter clearer.

**Example 6.1.1.** Let us consider the following recursive definition

$$q(x) \overset{\text{def}}{=} \text{tell}(y = 1) \parallel \text{next (local } y \text{) } (q(x) \parallel \text{when } y = 1 \text{ do } P).$$

In the case of dynamic scoping, an outside invocation $q(x)$ causes the execution of $P$ in the second time interval. The reason is that (local $y$) binds the $y$ resulting from the unfolding of the $q(x)$ inside the definition’s body\(^1\). In the case of static scoping, (local $y$) does not bind the $y$ of the unfolding of $A$ because such a $y$ is intuitively a “global” variable, and hence $P$ will not be executed. \hfill \Box

We shall avoid the kind of dynamic scoping mentioned above by requiring $fv(P_q) \subseteq \{x\}$, if a recursive call of $q$ occurs within $P_q$ in the lexical-scope of local process. For clarity, we shall then sometimes write

$$q(x)[y_1, \ldots, y_n] \overset{\text{def}}{=} P_{q},$$

to explicitly mention that variables $y_1, \ldots, y_n$ may occur free in $P_q$.

### 6.1.1 The Encoding.

Let us first to introduce some notation. Given $q(x) \overset{\text{def}}{=} P_q$, we will use $q, q\text{arg}$ to denote any two variables not in $fv(P_q)$. We use call($x$) as abbreviation of the constraint $x = 1$. Let $x \leftarrow t$ be an abbreviation of process

$$\sum_v \text{when } t = v \text{ do } \text{tell}(x = v).$$

Intuitively, $x \leftarrow t$ denotes the persistent assignment of $t$’s fixed value, say $v$, to $x$. Furthermore, let $\widehat{P}$ the process obtained by replacing in $P$ any call $q(t)$ with

$$\text{tell(call}(q)\text{)} \parallel \text{tell}(q\text{arg } t).$$

The idea is that such a replacement will tell that there is a call of $q$ with argument $t$ and excite a new copy of $P_q$.

The process corresponding to definition of $q(x)$, denoted as $^*q(x) \overset{\text{def}}{=} P_q^*$, is:

$$!(\text{when } \text{call}(q) \text{ do } (\text{local } x \leftarrow q\text{arg }) \parallel \widehat{P_q}).$$

The intuition is that whenever the process $q$ is called with argument $q\text{arg}$, the local $x$ is assigned the argument’s value so it can be used by $q$’s body $\widehat{P_q}$.

---

\(^1\)Just as in the CCS definition $A = a.0 \parallel \tau.(A \parallel \overline{a}.P)\backslash a$, the process $\overline{a}.P$ can communicate with the $a.0$ resulting from the unfolding of $A$, and then execute $P$. 
6.1. Recursion.

Finally, we consider the calls $q(t)$ in other processes. Each such a call is replaced by

$$(\text{local } q \ q_{\text{arg}}) \ (\tau q(x) \overset{\text{def}}{=} P_q \ | \ \hat{q}(t)), $$

which we shall denote by $\tau q(t)$\textsuperscript{−1}. The local declarations are needed to avoid interference with other recursive calls.

The encoding of recursion via replication above is slightly more complex than that of the $\pi$-calculus. The additional complexity arises from our desire of modeling directly call-by-value instead of call-by-name as this simplifies the presentation of the applications in this chapter.

Furthermore, our encoding also generalizes easily to the case of definition with arbitrary number of parameters of the form $q(\vec{x}) = P_q$ where $\vec{x}$ is a vector, possibly empty, of variables. In this chapter we will also use parameterless recursion, thus we ought to give the encoding for this particular the case. So, given a definition $q \overset{\text{def}}{=} P_q$ we define its encoding as $\tau q \overset{\text{def}}{=} P_q \ | \ \hat{q}$, where in this case $\hat{P}_q$ results from replacing in $P_q$ each call $q$ with tell(call($q$)). The calls to definition $q$ in other process should be replaced by the process $\tau q = (\text{local } q \ (\tau q \overset{\text{def}}{=} P_q \ | \ \hat{q})$.

We finish this section by introducing some convenient notation.

**Convention 6.1.2 (Notation for Recursion).** In the rest of this chapter we shall take the liberty of confusing the definitions $q(\vec{x}) \overset{\text{def}}{=} P_q$ and calls $q(\vec{v})$ with their corresponding ntcc translations $\tau q(\vec{v}) \overset{\text{def}}{=} P_q \ | \ \hat{q}$ and $\tau q(\vec{v}) \overset{\text{def}}{=} ! (\text{when call}(q) \ \text{do } \hat{P}_q)$, where in this case $\hat{P}_q$ results from replacing in $P_q$ each call $q$ with tell(call($q$)). The calls to definition $q$ in other process should be replaced by the process $\tau q = (\text{local } q \ (\tau q \overset{\text{def}}{=} P_q \ | \ \hat{q})$.

### 6.1.2 Proof Principles for Recursion

Here we state two results which gives us a proof principle for proving temporal properties of recursive definitions. The lemma below states a property that one would expect of recursive calls, i.e., if $B$ is satisfied by $q'$s body then $B[v/x]$ should be satisfied by $q(t)$ provided that $t = v$.

**Lemma 6.1.3 (Basic Temporal Property of Recursion).** Given a definition $q(x) \overset{\text{def}}{=} P_q$, suppose that $q, q_{\text{arg}}$ do not occur free in $B$ and that $P_q \vdash B$. Then for all $v \in \mathcal{D}$,

$$q(t) \vdash t = v \Rightarrow B[v/x].$$

**Proof.** From Theorem 5.2.6 we obtain the the strongest formula $F$ derivable for $q(t)$ (i.e., $\tau q(t)$\textsuperscript{−1}):

$$F = \exists q, q_{\text{arg}} (\text{call}(q) \hat{\land} q_{\text{arg}} = t \hat{\land} (\text{call}(q) \Rightarrow \exists x (A \hat{\land} H))$$

where $A$ is the strongest-formula derivable for $P_q$ (i.e., $\hat{P}_q$) and $H$ is the formula $\forall_w (q_{\text{arg}} = w \hat{\land} x = w) \forall_w \hat{\land} x = w$. From Definition 5.2.4, $A \Rightarrow B$. Let $G$ be the formula obtained from replacing $A$ with $B$ in $F$. Thus, $F \Rightarrow G$. From the Consequence Rule LC\text{CONS} in the proof system, we get $q(t) \vdash G$. Now let
\( v \) an arbitrary value in \( \mathcal{D} \). By observing that \( q \) does not occur free in \( B \) and some manipulations we verify that

\[
G \Rightarrow G' = \tilde{\tilde{\tau}}_{q \mathit{arg}}(q \mathit{arg} = t \land \tilde{\tilde{\tau}}_{x}(B \land H)).
\]

From Rule LCONS we get \( q(t) \vdash G' \). By observing that \( q \mathit{arg} \) does not occur free in \( B \), we verify that \( G' \Rightarrow (t = v \Rightarrow (B[v/x])) \). By applying LCONS again, we obtain the desired result, i.e., \( q(t) \vdash t = v \Rightarrow B[v/x] \).

For parameterless recursion we have the following version of the above lemma.

**Lemma 6.1.4 (Property of Parameterless Recursion).** Given a definition \( q \overset{\text{def}}{=} P_q \), suppose that \( q \) does not occur free in \( B \) and \( P_q \vdash B \). Then \( q \vdash B \).

**Proof.** Similar to the proof of Lemma 6.1.3

\[ \square \]

### 6.2 Cell Example

Cells provide a basis for the specification and analysis of mutable and persistent data structures. Here we illustrate how nttcc can be used to express and analyze cell behavior. We shall use the inference system for proving assertions of the form \( P \vdash A \). However, we shall first use the alternative method given by Corollary 5.2.7: determining whether \( \text{stf}(P) \Rightarrow A \).

Let us assume that the signature of the underlying constraint system is extended with an unary predicate symbol \( \text{change} \). A *mutable cell* \( x:(v) \) can be viewed as a structure \( x \) which has a current value \( v \) and which can, in the future, be assigned a new value.

\[
x: (z) \overset{\text{def}}{=} \text{tell}(x = z) \parallel \text{unless } \text{change}(x) \text{ next } x: (z)
\]

Intuitively, definition \( x:(z) \) represents a cell \( x \) whose value is \( z \) and it will be the same in the next time interval unless it is to be changed next (i.e., \( \text{change}(x) \)).

Below we give a temporal property which states the invariant behavior, or persistent nature if you wish, of a cell: If it satisfies \( A \) now, it will satisfy \( A \) next unless it is changed.

**Proposition 6.2.1 (Temporal Cell Invariance).** For all \( v \in \mathcal{D} \),

\[
x: (v) \vdash (A \land \lnot \exists x \text{change}(x)) \Rightarrow \diamond A.
\]

In the proof of the above proposition we illustrate how to prove an assertion of the form \( P \vdash A \) by determining the alternative provided by Corollary 5.2.7 rather than the inference system.
Proof of Proposition 6.2.1. From Corollary 5.2.7 it follows that is sufficient to prove the validity of $stf(x : (v)) \Rightarrow (A \land \lnot \text{change(x)}) \Rightarrow \Diamond A$ which is logically equivalent to the formula $(stf(x : (v)) \Rightarrow (A \land \lnot \text{change(x)})) \Rightarrow (stf(x : (v)) \Rightarrow \Diamond A)$.

Let us assume that $stf(x : (v)) \Rightarrow (A \land \lnot \text{change(x)})$. We want to conclude $stf(x : (v)) \Rightarrow \Diamond A$. One can verify that

$$(stf(x : (v)) \land \lnot \text{change(x)}) \Rightarrow \Diamond stf(x : (v)).$$

(6.2)

From the assumption $stf(x : (v)) \Rightarrow A$, thus from Equation 6.2

$$(stf(x : (v)) \land \lnot \text{change(x)}) \Rightarrow \Diamond A.$$  \hspace{1cm} (6.3)

But from the assumption $stf(x : (v)) \Rightarrow \lnot \text{change(x)}$, hence from Equation 6.3 we obtain $stf(x : (v)) \Rightarrow \Diamond A$ as wanted. \hfill \Box

Let us now introduce a generic operation on cells. Definition $\text{exch}_g[x, y]$ below represents an exchange operation in the following sense: if $v$ is $x$'s current value then $g(v)$ and $v$ will be the next values of $x$ and $y$ respectively.

$$\text{exch}_g[x, y] \overset{\text{def}}{=} \sum_v \text{when } x = v \text{ do (tell(change(x))) \| tell(change(y))}$$

$$\text{next } (x : (g(v)) \| y: (v)) ) \)$$

The following proposition describes the exchange operation in temporal logic terms:

Proposition 6.2.2 (Temporal Property of Exchange). For every $v \in D$,

$$\text{exch}_g[x, y] \vdash x = v \Rightarrow \Diamond (x = g(v) \land y = v).$$

We shall now describe in full detail the proof of the above proposition to illustrate the inference system (see Table 5.1) at work. The abbreviations Lem., Pr., and Eq. in the derivations below stands for “Lemma”, “Proposition” and “Equation”, respectively.

Proof of Proposition 6.2.2. Consider definition $x: (z) \overset{\text{def}}{=} P_x$: where $P_x$: has the form $\text{tell}(x = z) \parallel \text{unless change(x) next } x: (z)$. We conclude

$$P_x: \vdash x = z.$$ \hspace{1cm} (6.4)

from the derivation

$$\text{tell}(x = z) \vdash x = z \hspace{1cm} \text{LTELL}$$

$$P_x: \vdash x = z \hspace{1cm} \text{Lem.5.2.9(3)}$$

For $y: (z) \overset{\text{def}}{=} P_y$: with $P_y: = \text{tell}(y = z) \parallel \text{unless change(y) next } y: (z)$, we obtain a similar derivation of

$$P_y: \vdash y = z.$$ \hspace{1cm} (6.5)
By applying Lemma 6.1.3 to (6.4) and (6.5) we conclude that for arbitrary \( g, w \)
\[
x : (g(w)) \vdash x = g(w) \text{ and } y : (w) \vdash y = w.
\] (6.6)

Consider now \( \text{exch}_g[x, y] \overset{\text{def}}{=} Q = \sum_w \text{when } x = w \text{ do } Q_w \) where each
\( Q_w \) takes the form \( R \parallel \text{next} (x : (g(w)) \parallel y : (w)) \) with \( R \) being the process
\( \text{tell} (\text{change}(x)) \parallel \text{tell} (\text{change}(y)) \). We conclude
\[
Q \vdash (x = v \Rightarrow \circ(x = g(v) \wedge y = v)).
\] (6.7)

from the following derivation:

\[
\begin{aligned}
x : (g(w)) & \vdash x = g(w) \quad \text{Eq. (6.6)} \\
y : (w) & \vdash y = w \quad \text{Eq. (6.6)} \\
x : (g(w)) \parallel y : (w) & \vdash x = g(w) \wedge y = w \quad \text{LPAR} \\
\forall w \in D \quad Q_w & \vdash \circ(x = g(w) \wedge y = w) \quad \text{LNEXT} \\
Q & \vdash \bigwedge_{w \in D} (x = w \Rightarrow \circ(x = g(w) \wedge y = w)) \quad LSUM \\
Q & \vdash (x = v \Rightarrow \circ(x = g(v) \wedge y = w)) \quad LCONS
\end{aligned}
\]

From Equation (6.7) and Lemma 6.1.4 we obtain
\[\text{exch}_g[x, y] \vdash x = v \Rightarrow \circ(x = g(v) \wedge y = v)\]
as wanted. \( \square \)

Let us now consider a simple example of cell mutability. Often in exchange operations \( \text{exch}_g[x, y] \) we use functions \( g \) that always return the same value \( \text{i.e. constants}. \) For simplicity, we shall take the liberty of using that value as its symbol.

**Example 6.2.3.** The execution of \( x : (3) \parallel y : (5) \parallel \text{exch}_7[x, y] \) will give us in the next time unit the cells \( x \) and \( y \) with values 7 and 3, respectively. In fact, we get the following derivation

\[
\begin{aligned}
x : (3) & \vdash x = 3 \quad \text{Eq. (6.4)} \\
x : (3) \parallel y : (5) & \vdash x = 3 \quad \text{Lem. 5.2.9 (3)} \\
\text{exch}_7[x, y] & \vdash x = 3 \Leftrightarrow \circ(x = 7 \wedge y = 3) \quad \text{Pr. 6.2.2} \\
x : (3) \parallel y : (5) \parallel \text{exch}_7[x, y] & \vdash x = 3 \Rightarrow \circ(x = 7 \wedge y = 3) \quad \text{LPAR} \\
x : (3) \parallel y : (5) \parallel \text{exch}_7[x, y] & \vdash \circ(x = 7 \wedge y = 3) \quad \text{LCONS}
\end{aligned}
\]

Assignments, increasing and decreasing operations are typical cell operations. The assignment of \( v \) to a cell \( x \), written \( x := v \), can then be encoded as \( \text{local } y \text{ in } \text{exch}_y[x, y] \) where the local variable \( y \) is used as dummy variable (cell). Similarly, we can encode instructions \( x := x + 1 \) and \( x := x - 1 \) by using dummy variables, and the \textbf{succ} and \textbf{prd} functions. We shall use all these operations in our following application examples.
6.3 RCX Robots: The Zigzagging Example

An RCX is a programmable, controller-based LEGO® brick used to create autonomous robotic devices [LP99]. Zigzagging [Fre99] is a task in which an (RCX-based) robot can go either forward, left, or right but (1) it cannot go forward if its preceding action was to go forward, (2) it cannot turn right if its second-to-last action was to go right, and (3) it cannot turn left if its second-to-last action was to go left.

In order to model the above task, without over-specifying it, we use guarded choice. We use cells $a_1$ and $a_2$ to “look back” one and two time units, respectively. We use three distinct constants $f,r,l \in D - \{0\}$ and extend the signature with the predicate symbols $\text{forward, right, left}$.

\[
\begin{align*}
  GoF & \overset{\text{def}}{=} \text{exch}_{1}[a_1,a_2] \ || \ \text{tell(forward)} \\
  GoR & \overset{\text{def}}{=} \text{exch}_{2}[a_1,a_2] \ || \ \text{tell(right)} \\
  GoL & \overset{\text{def}}{=} \text{exch}_{3}[a_1,a_2] \ || \ \text{tell(left)} \\
  \text{Zigzag} & \overset{\text{def}}{=} !() \quad \text{when} \quad (a_1 \neq f) \ \text{do} \quad GoF \\
 & \quad + \ \text{when} \quad (a_2 \neq r) \ \text{do} \quad GoR \\
 & \quad + \ \text{when} \quad (a_2 \neq l) \ \text{do} \quad GoL \\
  GoZigzag & \overset{\text{def}}{=} a_1; (0) \ || \ a_2; (0) \ || \ \text{Zigzag}.
\end{align*}
\]

Initially cells $a_1$ and $a_2$ contain neither $f,r$ nor $l$. After a choice is made according to (1), (2) and (3), it is recorded in $a_1$ and the previous one moved to $a_2$.

One can verify that

\[ GoZigzag \models \Box (\Diamond \text{right} \land \Diamond \text{left}). \quad (6.8) \]

thus stating that the robot indeed goes right and left infinitely often. It is easy to see that we only need locally independent and restricted choice to express $GoZigzag$ (Definition 4.3.4). Therefore, it follows from the completeness result (Corollary 5.2.12) and Theorem 4.3.10 that there is actually a proof of the assertion in Equation 6.8 in our inference system. More precisely,

**Proposition 6.3.1 (A Zigzag Temporal Property).**

\[ GoZigzag \models \Box (\Diamond \text{right} \land \Diamond \text{left}). \]

The application in the following chapter will make use of process $GoZigzag$ above to model the unpredictable moves of a given agent.

6.4 Multi-Agent Systems: The Pursuit Example

The Predator/Prey (or Pursuit) game [BJD86] has been studied using a wide variety of approaches [HS96] and it has many different instantiations that can be used to illustrate different multi-agent scenarios [SV00]. As the Zigzagging
example, instances of the Predator/Prey game have been modeled using autonomous robots [NF98]. Here we model a simple instance of this game.

The predators and prey move around in a discrete, grid-like toroidal world with square spaces; they can move off one end of the board and come back on the other end. Predators and prey move simultaneously. They can move vertically and horizontally in any direction. In order to simulate fast but not very precise predators and a slower but more maneuverable prey we assume that predators move two squares in straight line while the prey moves just one.

The goal of the predators is to “capture” the prey. A capture position occurs when the prey moves into a position which is within the three-squares line of a predator current move; i.e. if for some of the predators, the prey current position is either the predator current position, the predator previous position, or the square between these two positions. This simulates the prey deadly moving through the line of attack of a predator.

For simplicity, we assume that initially the predators are in the same row immediately next to each other, while the prey is in front of a predator (i.e., in the same column, above this predator) one square from it. The prey’s maneuver to try to escape is to move in an unpredictable zigzagging around the world. The strategy of the predators is to co-operate to catch the prey. Whenever one of the predators is in front of the prey it declares itself as the leader of the attack and the other becomes its support. Therefore depending on the moves of the prey the role of leader can be alternated between the predators. The leader moves towards the prey, i.e. if it sees the prey above it then it moves up, if it sees the prey below it then it moves down, and so on. The support predator moves in the direction the leader moves, thus making sure it is always next to leader.

In order to model this example we extend the signature with the predicates symbols right, left, up, down for \( i \in \{0, 1\} \). For simplicity we assume there are only two predators \( Pred_0 \) and \( Pred_1 \). We use the cells \( x_i, y_i \) and cells \( x, y \) for representing the current positions of predator \( i \) and the prey, respectively, in an \( n \times n \) matrix (we assume that \( n = 2^k \) for some \( k > 1 \)) representing the world. We also use the primed version of these cells to keep track of corresponding previous positions and cell \( l \) to remember which predator is the current leader.

We can now formulate the capture condition. Predator \( i \) captures the prey with a horizontal move iff

\[
x_i' = x = x_i \land \left( (y_i = y_i' - 2 \land (y = y_i' \lor y = y_i' - 1 \lor y = y_i' - 2)) \lor \\
(y_i = y_i' + 2 \land (y = y_i' \lor y = y_i' + 1 \lor y = y_i' + 2)) \right)
\]

and with a vertical move iff

\[
y_i' = y = y_i \land \left( (x_i = x_i' - 2 \land (x = x_i' \lor x = x_i' - 1 \lor x = x_i' - 2)) \lor \\
(x_i = x_i' + 2 \land (x = x_i' \lor x = x_i' + 1 \lor x = x_i' + 2)) \right).
\]

We define \texttt{capture} \( i \) as the conjunction of the two previous constraints.

The process below models the behavior of the prey. The prey moves as in the Zigzagging example. Furthermore, the values of cells \( x, y \) and \( x', y' \) are updated according to the zigzag move (e.g., if it goes right the value of \( x \) is
increased and \( x' \) takes \( x \)'s previous value).

\[
\text{Prey} \overset{\text{def}}{=} \text{GoZigzag} \parallel ! (\text{ when forward do } \text{exch}_{\text{pred}}[y, y'] \\
+ \text{ when right do } \text{exch}_{\text{succ}}[x, x'] \\
+ \text{ when left do } \text{exch}_{\text{pred}}[x, x'])
\]

The process \( \text{Pred}_i \) with \( i \in \{0, 1\} \) models the behavior of predator \( i \). The operator \( \oplus \) denotes binary summation.

\[
\text{Pred}_i \overset{\text{def}}{=} ! (\text{ when } x_i = x \quad \text{do } (l := i \parallel \text{Pursuit}_i) \\
+ \text{ when } l = i \land x_i = x \quad \text{do } \text{Pursuit}_i \\
+ \text{ when } l = i \oplus 1 \land x_i \neq x \quad \text{do } \text{Support}_i)
\]

Thus whenever \( \text{Pred}_i \) is in front of the prey (i.e., \( x_i = x \)) it declares itself as the leader by assigning \( i \) to the cell \( l \). Then it runs process \( \text{Pursuit}_i \) defined below and keeps doing it until the other predator \( \text{Pred}_i \oplus 1 \) declares itself the leader. If the other process is the leader then \( \text{Pred}_i \) runs process \( \text{Support}_i \) defined below.

The process \( \text{Pursuit}_i \), whenever the prey is above of corresponding predator (\( y_i < y \land x_i = x \)), tells the other predator that the move is to go up and increases by two the contents of \( y_i \) while keeping in cell \( y_i' \) the previous value. The other cases which correspond to going left, right and down can be described similarly.

\[
\text{Pursuit}_i \overset{\text{def}}{=} \text{when } (y_i < y \land x_i = x) \text{ do } \\
(\text{exch}_{\text{succ}}[y_i, y_i'] \parallel \text{tell(up)}_i) \\
+ \text{when } (y_i > y \land x_i = x) \text{ do } \\
(\text{exch}_{\text{pred}}[y_i, y_i'] \parallel \text{tell(down)}_i) \\
+ \text{when } (x_i < x \land y_i = y) \text{ do } \\
(\text{exch}_{\text{succ}}[x_i, x_i'] \parallel \text{tell(right)}_i) \\
+ \text{when } (x_i > x \land y_i = y) \text{ do } \\
(\text{exch}_{\text{pred}}[x_i, x_i'] \parallel \text{tell(left)}_i)
\]

The process \( \text{Support}_i \) is defined according to the move decision of the leader. Hence, if the leader moves up (e.g. \( \text{up}_{i \oplus 1} \)) then the support predator moves up as well. The other cases are similar.

\[
\text{Support}_i \overset{\text{def}}{=} \text{when } \text{up}_{i \oplus 1} \text{ do } \\
(\text{exch}_{\text{succ}}[y_i, y_i'] \parallel \text{tell(up)}_i) \\
+ \text{when } \text{down}_{i \oplus 1} \text{ do } \\
(\text{exch}_{\text{pred}}[y_i, y_i'] \parallel \text{tell(down)}_i) \\
+ \text{when } \text{right}_{i \oplus 1} \text{ do } \\
(\text{exch}_{\text{succ}}[x_i, x_i'] \parallel \text{tell(right)}_i) \\
+ \text{when } \text{left}_{i \oplus 1} \text{ do } \\
(\text{exch}_{\text{pred}}[x_i, x_i'] \parallel \text{tell(left)}_i)
\]

We assume that initially \( \text{Pred}_0 \) is the leader and that it is in the first row in the middle column. The other predator is next to it in the same row. The prey is just above \( \text{Pred}_0 \). The process \( \text{Init} \) below specifies these conditions. Let \( p = n/2 \).

\[
\text{Init} \overset{\text{def}}{=} \prod_{i=0,1} (x_i : (p + i) \parallel y_i : (0) \parallel x_i' : (p + i) \parallel y_i' : (0)) \\
\parallel x : (p) \parallel y : (1) \parallel x' : (p) \parallel y_i' : (1) \parallel l : 0.
\]
Operationally, one can verify that

\[
\text{Init} \parallel \text{Pred}_0 \parallel \text{Pred}_1 \parallel \text{Prey} \models \Diamond (\text{capture}_0 \lor \text{capture}_1)
\]

thus stating that the predators eventually capture the prey under our initial conditions. Notice that only locally-independent and restricted choice constructs are needed to express this application. Hence, from the completeness result of the inference system (and Theorem 4.3.10,) we obtain the following property.

**Proposition 6.4.1 (A Temporal Property of Predators).**

\[
\text{Init} \parallel \text{Pred}_0 \parallel \text{Pred}_1 \parallel \text{Prey} \models \Diamond (\text{capture}_0 \lor \text{capture}_1).
\]

It is worth noticing that in the case of one single predator, say Pred\(_0\), the prey may sometimes escape under the same initial conditions which can be expressed as \(\text{Init} \parallel \text{Pred}_0 \parallel \text{Prey} \models \Diamond \text{capture}_0\). A similar situation occurs if the predators were not allowed to alternate the leader role.

### 6.5 Musical Applications

In the last decade several formalisms have been proposed to account for musical structures and the operations used to construct and transform them [BBDC98, BS93, PRC96]. We can regard music performance and composition as a complex task of defining and controlling interaction among concurrent activities. In [ADQ'01], PiCO, a concurrent processes calculus integrating constraints and objects was proposed. Musical applications are programmed in a visual language having this calculus as its underlying model. Since there is no explicit notion of time in PiCO some musical examples, in particular those involving time and synchronization, are difficult to express. In this section we model two of those examples.

In the following examples we shall used the derived operators \(*_{[m,n]}P\) and \(\lbrack m,n\rbrack P\) introduced in Section 2.2.8. Recall that \(*_{[m,n]}P\) means that \(P\) is eventually active between the next \(m\) and \(m + n\) time units, while \(\lbrack m,n\rbrack P\) means that \(P\) is always active between the next \(m\) and \(m + n\). We shall also use the derived construct

\[
W_{(c,P)} \overset{\text{def}}{=} \text{when } c \text{ do } P \parallel \text{unless } c \text{ next } W_{(c,P)}.
\]

This construct waits until \(c\) holds and then executes \(P\). Instead of \(W_{(c,P)}\), we shall use the more readable notation \text{wait } c \text{ do } P.

#### 6.5.1 Controlled Improvisation.

This example models a controlled improvisation musical system. Such a system can be described as follows: There is a certain number \(m\) of musicians (or voices), each playing blocks of three notes. Each of them is given a particular pattern (i.e., a list) of allowed delays between each note in the block. The musician can freely choose any permutation of his pattern. For example, given a pattern \(p = [4,3,5]\) a musician can play his block with spaces of 5 then 4
and then 3 between the notes. Once a musician has finished playing his block of three notes, he must wait for a signal of the conductor telling him that the others musicians have also finished their respective blocks. Only after this he can start playing a new block. The exact time in which he actually starts playing a new block is not specified, but it is constrained to be no later than the sum of the durations of all patterns. For example, for three musicians and patterns \( p_1 = [3, 2, 2] \), \( p_2 = [4, 3, 5] \) and \( p_3 = [3, 3, 4] \) no delay between blocks greater than 29 time units is allowed. The musicians keep playing this way until all of them play a note at the same time. After this, all the musicians must stop playing.

In order to model this example we assume that constant \( \text{sil} \in D \) represents some note value for silence. The process \( M_i, i \leq m \), models the activity of the \( i \)-th musician. When ready to start playing \( (\text{start}_i = 1) \), the \( i \)-th musician chooses a permutation \( (j,k,l) \) of his given pattern \( p_i \). Then, \( M_i \) spawns a process \( \text{Play}^i_{(j,k,l)} \), thus playing a note at time \( j \) (after starting), but not before, then at time \( j+k \) but not before, and finally at time \( j+k+l \). Constraint \( c_i[\text{note}_i] \) specifies some value for \( \text{note}_i \) different from \( \text{sil} \). After playing his block, the \( i \)-th musician signals termination by setting cell \( \text{flag}_i \) to 1. Furthermore, upon receiving the \( \text{go} = 1 \) signal, the \( i \)-th musician eventually starts a new block no later than \( \text{pdur} \) which is a constant representing the sum of the durations of all patterns.

\[
M_i \overset{\text{def}}{=} \!\text{when} (\text{start}_i = 1) \text{ do } \sum_{(j,k,l) \in \text{perm}(p_i)} ( \text{Play}^i_{(j,k,l)} ) \text{ next } j+k+l( \text{flag}_i := 1 \text{|| wait (go = 1) do } \\
\text{tell}[0,\text{pdur}](\text{start}_i = 1) ) )
\]

\[
\text{Play}^i_{(j,k,l)} \overset{\text{def}}{=} \\
\text{tell}[0,j-1](\text{note}_i = \text{sil}) \text{ next } j\text{tell}(c_i[\text{note}_i]) \text{ next } j\text{tell}(c_i[\text{note}_i]) \text{ next } j+k\text{tell}(c_i[\text{note}_i]) \text{ next } j+k+l\text{tell}(c_i[\text{note}_i])
\]

The \( \text{Conductor} \) process is always checking (listening) whether all the musicians play a note exactly at the same time \( \bigwedge_{i \in [1,m]} (\text{note}_i \neq \text{sil}) \). If this happens it sets the cell \( \text{stop} \), initially set to 0, to 1. At the same time, it waits for all flags to be set to 1, and then resets the flags and gives the signal \( \text{go} = 1 \) to all musician to start a new block, unless all of them have output a note at the same time (i.e., \( \text{stop} = 1 \)).

\[
\text{Conductor} \overset{\text{def}}{=} \\
\text{wait } \bigwedge_{i \in [1,m]} (\text{note}_i \neq \text{sil}) \text{ do } \text{stop} := 1 \text{|| } \!\text{when } \bigwedge_{i \in [1,m]} (\text{flag}_i = 1) \land (\text{stop} = 0) \text{ do } (\text{tell}(\text{go} = 1) \text{|| } \prod_{i \in [1,m]} \text{flag}_i := 0)
\]
Initially the \( m \) flag cells are set to 0, the \( M_i \) are given the start signal \( \text{start}_i = 1 \) and, as mentioned above, the cell \( \text{stop} \) is set to 0. The system (i.e., the performance) \( \text{System} \) is just the parallel execution (performance) of all the \( M_i \) musicians controlled by the \( \text{Conductor} \) process.

\[
\text{Init} \overset{\text{def}}{=} \prod_{i \in [1,m]} (\text{flag}_i : 0 \parallel \text{tell}(\text{start}_i = 1) \parallel \text{stop} : 0)
\]

\[
\text{System} \overset{\text{def}}{=} \text{Init} \parallel \text{Conductor} \parallel \prod_{i \in [1,m]} M_i
\]

The temporal logic and the proof system of \( \text{ntcc} \) can then be used to formally specify and prove termination properties for this system. For example, we may wonder whether the assertion

\[
\text{System} \models \Diamond \text{stop} = 1
\]

holds. This assertion expresses that the musicians eventually stop playing at all regardless their choices. We may also wonder whether there exists certain choices of musicians for which they eventually stops playing note at all. For proving this we can verify whether the assertion

\[
\text{System} \models \Box \text{stop} = 0
\]

does not hold, i.e., there is a run of the system for which at some time unit all the notes are different from \( \text{sil} \). Notice that process \( \text{System} \) can be expressed in the locally-independent fragment. We can then use inference system to reason about the above termination properties.

### 6.5.2 Rhythm Patterns

In this section we shall model synchronization of rhythm patterns in \( \text{ntcc} \). Let us first define a "metronome" process.

\[
M[\text{tick}, \text{count}, \delta] \overset{\text{def}}{=} \\
! (\text{when} (\text{count} \mod \delta = 0 \text{ do } \text{tick} := \text{tick} + 1 \parallel \text{count} := 0) \\
+ \text{when} (\text{count} \mod \delta > 0 \text{ do } \text{count} := \text{count} + 1)
\]

One could think of \( M[\text{tick}, \text{count}, \delta] \) as a process that "ticks" (by increasing \( \text{tick} \)) every \( \delta \) time units. This process could be controlled by the acceleration process:

\[
\text{Accel}[\text{signal}, \delta] \overset{\text{def}}{=} ! \text{when } \text{signal} = 1 \land \delta > 0 \text{ do } \delta := \delta - 1
\]

The process \( \text{Accel}[\text{signal}, \delta] \) can "speed up the ticks of \( M[\text{tick}, \text{count}, \delta] \)" by decreasing \( \delta \), if some other process, which we shall refer to as \( \text{Control}[\text{signal}] \), tells \( \text{signal} = 1 \).

We can now define the \( \text{Rhythm} \) process \( R_{(s,d,e)}[\text{tick}, \text{note}] \) which can be synchronized by \( M[\text{tick}, \text{count}, \delta] \) and thus possibly accelerated by \( \text{Control}[\text{signal}] \).
\[ R_{(s,d,e)}[\text{tick, note}] = \begin{cases} 
\text{when } s >= \text{tick} >= e \land (\text{tick} - \text{start}) \mod d = 0 \text{ do } \text{Pitch[note]} 
\end{cases} \]

The process \( R_{(s,d,e)}[\text{tick, note}] \) runs a certain \( \text{Pitch[note]} \) process, which outputs some pitch on \( \text{note} \), at every \( (d) \)uration-th tick, from the \( (s) \)tart-th tick to the \( (e) \)nd-th tick. Adding rhythms of two notes, two triplets and two quintuplets can then be defined by the system:

\[
\text{System} = \text{def} \quad (\text{local } \text{tick } \delta \ \text{count } \text{signal}) ( \\
\text{Init} \ || \ \text{Control[signal]} \ || \ M[\text{tick, count, } \delta] \ || \ \text{Accel(signal, } \delta) \\
\ || \ R_{(0,30,130)}[\text{tick, } \delta] \ || \ R_{(0,30,120)}[\text{tick, } \delta] \ || \ R_{(0,12,120)}[\text{tick, } \delta] )
\]

where \( \text{Init} = \text{tick} : 0 \ || \ \delta : 30 \ || \ \text{count} : 0. \)

Since the \( \text{Rhythm} \) processes depend on variables \( \text{tick} \) and \( \delta \), complex patterns of interactions of global and local speeds, such as metric modulations, can be modeled.

### 6.6 Summary

We have illustrated several application examples of the \texttt{ntcc} calculus, its logic and its proof system. Such examples included a basic form of call-by-value recursion and data-structures. Our main purpose was to illustrate how \texttt{ntcc} can be used to express the behavior and analyze properties of these two examples. We also illustrated how to use \texttt{ntcc} to model more complex examples involving robotic devices, multi-agent systems, and music applications. With these examples we also illustrated the expressiveness of the locally independent fragment of \texttt{ntcc} (for which, in Chapter 5, we proved completeness of the proof system).

The musical and multi-agent application were first presented in [RV01] and [NV02], respectively. The other applications were published in [PV01].
Chapter 7

Observable Behavior: Decidability and Congruence Issues

We see what we see because we miss all the finer details.
— Alfred Korzybski

In Chapter 3 we introduced notions of observation that abstract way of unimportant internal details (see Definition 3.5.3). Given a process $P$, the first such observation, called input-output behavior and denoted by $io(P)$, is the infinite sequences of input-output interactions in which $P$ can engage with an environment. The second, is the default output behavior $o(P)$ which focus on the output behavior of $P$ in the absence of an external environment. The third, called strongest-postcondition and denoted by $sp(P)$, focus on the output behavior in the presence of arbitrary environments.

The above behavioral observations induce equivalences which identify processes whose observable behavior follow the same pattern but whose internal behavior may be different. Namely, the input-output, default output, and strongest-postcondition equivalences, written as $\sim_{io}$, $\sim_{o}$ and $\sim_{sp}$, respectively (Definition 3.5.3). In this chapter, we shall study these observational equivalences and the relationship among them as well as their decidability.

This chapter is structured in three main sections. In the first section we state the relationship among the various equivalences and their induced congruences. In particular, we show that although $\sim_{io}$ is stronger than $\sim_{o}$, their corresponding induced congruences coincide for the star-free fragment of $\text{ntcc}$.

Then, in the second section we characterize $\sim_{io}$, $\sim_{sp}$ and the output congruence in terms of $\sim_{o}$ by identifying an interesting family of “distinguishing” contexts. For example, the (default) output congruence characterization states that $P$ and $Q$ are output congruent iff they are output equivalent under a particular context which, given $P$ and $Q$, can be effectively constructed. If the underlying set of constraints $C$ is finite, such context is universal, meaning that it is the same for any $P$ and $Q$. The other characterizations state analogous results.

Finally, in the third section, we show that $\sim_{o}$ is decidable for processes with a restricted form of nondeterminism. Namely, star-free processes in which local operators do not exhibit nondeterminism. The decidability result is obtained
by characterizing the output behavior in terms of finite-state automata over infinite sequences. From the decidability of $\sim_\alpha$, it will follow that the output and input-output congruences as well as $\sim_{io}$ and $\sim_{sp}$ are all decidable for such a restricted form of nondeterminism.

7.1 Process Equivalences and Induced Congruences

In Chapter 4 we discussed about the lack of compositionality of the strongest-postcondition arising from the combination between nondeterminism and locality. This already suggests that $\sim_{sp}$ is not a congruence (see Definition 3.3.3). From the full-abstraction result given in Corollary 4.3.17, however, we can conclude that $\sim_{sp}$ is congruence in the restricted sense expressed in the theorem below.

**Theorem 7.1.1 (SP Restricted Congruence).** The strongest postcondition equivalence $\sim_{sp}$ satisfies the following:

1. $\sim_{sp}$ is not a congruence.

2. $P \sim_{sp} Q$ if and only if for all context $C[\cdot]$, s.t. $C[P]$ and $C[Q]$ are locally-independent (Definition 4.3.4), we have $C[P] \sim_{sp} C[Q]$.

**Proof.** Item (2) is an immediate consequence of Corollary 4.3.17. Item (1) follows from the following counterexample:

Let $P = (\text{when } x = 1 \text{ do tell}(y = 1)) + (\text{when true do tell}(\text{false}))$ and $Q = \text{tell}(x = 1 \land y = 1)$. Processes $P$ and $Q$ have the same strongest postcondition (i.e., the set of all $c\alpha$ such that $c \models x = 1 \land y = 1$). Notice, however, the sequence $(x = 1 \land y = 1).\text{true}$ is quiescent for $(\text{local } x) Q$ but not for $(\text{local } x) P$. 

Unfortunately, because of the choice operator, the input-output equivalence $\sim_{io}$ and the default output $\sim_\alpha$ are not are not congruences either as shown by the following counter-example.

**Example 7.1.2.** Assume that $a, b, c$ are non-equivalent constraints such that $c \models b \models a$. Let

\[P = (\text{when true do tell}(a)) + (\text{when b do tell}(c))\]
\[Q = (\text{when true do tell}(a)) + (\text{when b do tell}(c)) + (\text{when true do (tell}(a) \parallel \text{when b do tell}(c)))\]

and let $R = \text{when a do tell}(b)$. We leave it to the reader to verify that we can distinguish $P$ from $Q$ if we make $R$ to interact with them, i.e. although $P \sim_{io} Q$ (and thus $P \sim_\alpha Q$) we have $R \parallel P \not\sim_\alpha R \parallel Q$ (and thus $R \parallel P \not\sim_{io} R \parallel Q$). Therefore $P \not\sim_\alpha Q$ and $P \not\sim_{io} Q$.

Consequently, we ought to consider the largest congruences $\sim_{io}$ and $\sim_\alpha$ included in $\sim_{io}$ and $\sim_\alpha$, respectively. We shall confine our attention, however,
7.1. Process Equivalences and Induced Congruences

to the star-free fragment of $\text{ncc}$. The relevance of such a fragment is that its transition system is finitely-branching (Theorem 3.4.8, Item (4)). As we shall clarify, it is not clear to what extent our results for the star-free fragment extend to the full language.

**Convention 7.1.3.** We shall often abuse the notation by using $\sim_{io}$, $\sim_o$ and $\sim_{sp}$ to denote also the input-output, (default) output and strongest-postcondition equivalence, respectively, of star-free processes only. It should be clear from the context, however, what each equivalence is denoting.

**Definition 7.1.4 (Observable Congruences).** The relations over star-free processes $\approx_w$ and $\approx_o$ are given by: $P \approx_{io} Q$ iff for every star-free context $C[.]$, $C[P] \approx_{io} C[Q]$, and $P \approx_o Q$ iff for every star-free context $C[.]$, $C[P] \approx_o C[Q]$.

In the next section we shall provide a characterization of the above congruences.

Let us conclude this section by looking at the relationship between the different equivalences. Recall that $\equiv$ denotes structural congruence (Definition 3.3.4). For technical purposes we consider the finite prefixes of the default output of processes. Let $o^i(P) = \{\alpha^i | \alpha \in o(P)\}$ where $\alpha^i$ is the $i$-th prefix of $\alpha$ and define $P \sim_{io} Q$ iff $o^i(P) = o^i(Q)$.

Obviously, relation $\sim_o$ is weaker than $\sim_{io}$, however, the corresponding congruences coincide.

**Theorem 7.1.5.** The equivalences $\equiv$, $\sim_{io}$, $\sim_o$ and $\sim_{sp}$ over star-free processes satisfy the following:

$$
\equiv \subset \approx_{io} \subset \sim_{io} \subset \approx_o \subset \sim_o = \bigcap_{n \in \omega} \sim_o^n .
$$

**Proof.** The first proper inclusion follows from the fact that the operational semantics is quotiented by $\equiv$; the second follows from Example 7.1.2. The third proper inclusion is trivial and left to reader to verify. The final equality follows from the fact that the transition system for star-free processes is finitely branching (Theorem 3.4.8, Item 4). Here we prove $\approx_{io} = \approx_o$. The case $\approx_{io} \subseteq \approx_o$ is trivial. We want to prove that $P \approx_o Q$ implies $P \approx_{io} Q$. Suppose that $P \approx_o Q$ but $P \not\approx_{io} Q$. Then there must exist a context $C[.]$ s.t $C[P] \not\approx_{io} C[Q]$. Consider the case $io(C[P]) \not\approx io(C[Q])$. Take an $\alpha = c_1c_2\ldots$ such that $(\alpha, \alpha') \in io(C[Q])$ but $(\alpha, \alpha') \not\in io(C[P])$. Because the transition system is finitely branching, by using the fan theorem\footnote{The fan theorem states that for every finitely branching tree, with all its branches being finite, there is an upper bound on the length of the branches.} we conclude that there must be a prefix of $\alpha'$ which differs from all other prefixes of sequences $\alpha''$ s.t. $(\alpha, \alpha'') \in io(C[P])$. Suppose that this is the $n$–th prefix. One can verify that for the context

$$
C'[.] = C[.] \parallel \prod_{i \leq n} \text{next}^i \text{tell}(c_i),
$$

$io(C'[P]) \not\approx io(C'[Q])$. This contradicts our assumption $P \approx_o Q$. The case $io(C[Q]) \not\approx io([P])$ is symmetric. Therefore $P \not\approx_o Q$ as required. \hfill $\square$

7.2 Characterization of Behavioral Equivalences

We next investigate the type of contexts \( C[\cdot] \) in \( \text{ntcc} \) needed to verify the various equivalences. We shall work with the relation \( \approx_o \) since, as we shall see later, they provide enough information to characterize the other equivalence relations.

**Notation 7.2.1.** Throughout the chapter we shall often use the following notation on transitions:

1) \( P \rightarrow Q \) \quad iff \quad \langle P, c \rangle \rightarrow \langle Q, d \rangle, \text{ for some } c, d.

2) \( P \Rightarrow Q \) \quad iff \quad P \xrightarrow{(c,d)} Q, \text{ for some } c, d.

3) \( P \xrightarrow{c} Q \) \quad iff \quad P \xrightarrow{(\text{true},c)} Q.

4) \( P \xrightarrow{\alpha} \omega \) \quad iff \quad P \xrightarrow{(\text{true}^*,\alpha)} \omega.

We shall start by giving a result which simplifies the kind of contexts needed to verify output congruence. Such a result appeals to the following proposition which allows us to approximate the behavior of \( !P \).

**Proposition 7.2.2.** For all star-free \( P \) and \( Q \), for all \( n \geq 0 \), we have

\[
Q \parallel !P \approx^n_0 Q \parallel \prod_{i \leq n} \text{next}^i P.
\]

**Proof.** By induction on \( n \). \(\)\)

The lemma below states that in order to verify \( \approx_o \) it is sufficient to consider parallel contexts only.

**Lemma 7.2.3.** \( P \approx_o Q \) \quad iff \quad \text{for all } R, R \parallel P \approx R \parallel Q.

**Proof.** The “only if” direction is trivial. Consider the “if” direction. Suppose that for all \( R, R \parallel P \approx R \parallel Q \). It suffices to prove that for all contexts \( C[\cdot] \), \( R \parallel C[P] \approx C[R] \) for an arbitrary \( R \). The proof proceeds by induction on the structure of \( C \). Here we outline the proof of the next and replication context cases. The other cases are trivial.

Consider \( C = \text{next}[\cdot] \). We have \( R \parallel \text{next} P \xrightarrow{c} R' \parallel P \iff R \xrightarrow{c} R' \). Similarly, \( R \parallel \text{next} Q \xrightarrow{c} R' \parallel Q \iff R \xrightarrow{c} R' \). Thus, the result follows immediately from the initial assumption.

Consider \( C = ! [\cdot] \). From the Prop. 7.2.2 for all \( n, R \parallel !P \approx^n_0 R \parallel \prod_{i \leq n} \text{next}^i P \) and \( R \parallel !Q \approx^n_0 R \parallel \prod_{i \leq n} \text{next}^i Q \). With the help of Theorem 7.1.5 \( ( \approx_o = \bigcap_{n \in \omega} \approx^n_0 ) \) we conclude that \( R \parallel !P \approx R \parallel !Q \) if for all \( n \geq 0 \), \( R \parallel !P \approx^n_0 R \parallel !Q \). The result now follows from the next and parallel cases. \(\)

The above result provides a little simplification of the kind of contexts needed to verify output congruence. In the next section, however, we shall show that there is indeed a more effective way of characterizing output congruence, and indeed the other equivalences, via output equivalence.

We find it convenient to consider first the case in which the underlying set of constraints \( C \) is finite. Then we shall turn to the general case.
7.2.1 The Finite Case

In this section we shall introduce contexts that, assuming that \( C \) is finite, can be used to distinguish via output equivalence any two processes that are different under \( \sim_{sp}, \sim_{io} \) or \( \approx_{o} \) (and thus \( \approx_{io} \)). We shall then refer to such contexts as being universal.

**Universal Contexts**

Notice that the replicated process \( ! \sum_{c \in S} tell(c) \) can provide any input constraint that by an environment whose inputs are restricted to \( S \).

**Definition 7.2.4 (Context \( U_{sp} \)).** Let \( S \subseteq_{fin} C \). Define \( U_{sp}(S) \) to be the context

\[
( ! \sum_{c \in S} tell(c) ) \parallel [\cdot] 
\]

From the above observation it not difficult to see that context \( U_{sp}(C) \) can be used to distinguish via output equivalence any two process with a different postcondition. More precisely,

**Proposition 7.2.5 (SP Universal Context).** Suppose that \( C \) is finite. Then

\[
P \sim_{sp} Q \iff U_{sp}(C)[P] \sim_{o} U_{sp}(C)[Q].
\]

**Proof.** The “if” direction follows from the fact that (the summation in) the universal context \( U_{sp}(C) \) can provide any input given by the environment. The “only if” direction follows trivially from the fact that the environment can provide any constraint that the summation in the universal context can choose to tell. Here we illustrate the “if” direction.

Suppose that \( U_{sp}(C)[P] \sim_{o} U_{sp}(C)[Q] \). Consider the case \( sp(P) \subseteq sp(Q) \). Suppose that \( \alpha = c_1, c_2, \ldots \in sp(P) \). We want to prove \( \alpha \in sp(Q) \).

Let \( R = ! \sum_{c \in C} tell(c) \). Since \( \alpha \in sp(P) \), then from Theorem 3.5.2 it follows that

\[
P = P_1 \xrightarrow{(c_1, c_1)} P_2 \xrightarrow{(c_2, c_2)} \ldots. \tag{7.1}
\]

Notice that

\[
R \xrightarrow{\text{true}, c_1} R \xrightarrow{\text{true}, c_2} \ldots. \tag{7.2}
\]

Then from Equations 7.1 and 7.2 we can verify that

\[
R \parallel P_1 \xrightarrow{\text{true}, c_1} R \parallel P_2 \xrightarrow{\text{true}, c_2} \ldots.
\]

Therefore \( \alpha \in o(R \parallel P) \) and then from the initial assumption, \( \alpha \in o(R \parallel Q) \). By definition \( \alpha \in sp(R \parallel Q) \), and hence by Theorem 3.5.2 \( \alpha \) is quiescent for \( R \parallel Q \) (i.e., \( R \parallel Q \) can run on \( \alpha \) without adding any information). It is easy to prove that if \( \alpha \) is quiescent for \( R \parallel Q \) then it is quiescent for \( Q \) as well. By Theorem 3.5.2 \( \alpha \in sp(Q) \) as wanted. The case \( sp(Q) \subseteq sp(P) \) is symmetric. \( \square \)
Similarly, under the finiteness assumption of $C$, we can construct a context that can be used to distinguish via output equivalence any two processes with different input-output behavior.

Let us assume that the underlying constraint system signature is extended with constant predicates $p_c$ for each $c \in C$. Furthermore, such predicates are decreed “private” in the sense that they must not occur in any input constraint or context other than the contexts $U_{io}(S)$ defined below. The assertion $S \subseteq_{fin} S'$ holds iff $S$ is a finite subset of $S'$.

**Definition 7.2.6 (Context $U_{io}$).** Let $S \subseteq_{fin} C$. Define $U_{io}(S)$ to be the context
\[(\bigcup_{c \in S} (\text{tell}(c) \ || \ \text{tell}(p_c)))) \parallel .\]

We shall use each predicate $p_c$ as a witness of choice of $\text{tell}(c)$ made by the summation in context $U_{io}(S)$. Thus, if $U_{io}(S)[P] \xrightarrow{\text{true}, d \wedge P_k} U_{io}(S)[Q]$ then we can infer that the summation chose $\text{tell}(c)$ for execution. This is because, from our assumption, $p_c$ is not related by entailment to any other constraint in $U_{io}[P]$. Then it follows that $P \xrightarrow{(c, d)} Q$. More precisely,

**Proposition 7.2.7 (IO Universal Context).** Suppose that $C$ is finite. Then for every star-free $P, Q$,
\[P \sim_{io} Q \text{ iff } U_{io}(C)[P] \sim_o U_{io}(C)[Q].\]

**Proof.** Both directions follows from facts similar to those expressed in the proof of Proposition 7.2.5. Here we illustrate the “if direction”.

Suppose that $U_{io}(C)[P] \sim_o U_{io}(C)[Q]$. Consider the case $io(P) \subseteq io(Q)$. Let $\alpha = c_1, c_2, \ldots$ and $\alpha' = c'_1, c'_2, \ldots$ such that $(\alpha, \alpha') \in io(P)$. We want to prove $(\alpha, \alpha') \in io(Q)$.

Let $R = \bigcup_{c \in C} (\text{tell}(c) \ || \ \text{tell}(p_c))$. Notice that
\[R \xrightarrow{\text{true}, c_1 \wedge P_1} R \xrightarrow{\text{true}, c'_1 \wedge P_2} \ldots\]

Thus, since $(\alpha, \alpha') \in io(P)$, we can verify that that, for $P = P_1$,
\[R \parallel P_1 \xrightarrow{\text{true}, c'_1 \wedge P_1} R \parallel P_2 \xrightarrow{\text{true}, c'_2 \wedge P_2} \ldots\]

From the initial assumption, it follows that for $Q = Q_1$,
\[R \parallel Q_1 \xrightarrow{\text{true}, c'_1 \wedge P_1} R \parallel Q_2 \xrightarrow{\text{true}, c'_2 \wedge P_2} \ldots\]

From our assumption about the privacy of the $p_c$’s predicates, we conclude that, since $p_c$ is told at each time $i$, $R$ chose $\text{tell}(c_i)$ at each time $i$. We can then prove that
\[Q_1 \xrightarrow{(c_1 \wedge P_{1}, c'_1 \wedge P_1)} Q_2 \xrightarrow{(c_2 \wedge P_{2}, c'_2 \wedge P_2)} \ldots\]

Again from the privacy of the $p_c$’s predicates it follows that
\[Q_1 \xrightarrow{(c_1, c'_1)} Q_2 \xrightarrow{(c_2, c'_2)} \ldots\]
as wanted. The case $io(Q) \subseteq io(P)$ is symmetric. \qed
7.2. Characterization of Behavioral Equivalences

We now consider the case of providing an universal distinguishing context for the output congruence $\approx_o$. Our plan is to provide a context that can simulate all possible interactions that a given process can have with others. We therefore may expect such a context to be somewhat more complex than the universal contexts for the strongest postcondition and input-output equivalences.

Consider $R \parallel P$ with $P$ and $R$ as in Example 7.1.2. By telling information, the process $P$ provides information which influences the evolution of $R$, i.e., the constraint $a$. Similarly, $R$ influences the evolution of $P$ by providing the constraint $b$. Thus asking $a$ and then telling $b$ is one possible interaction a process can have with $P$ while telling $a$ and then asking $b$ is a possible interaction a process can have with $R$. In general, interactions can be represented as strictly increasing and alternating sequences of ask and tell operations (see [SRP91]).

In the following we write $c \prec c'$ iff $c \models c'$ and $c \nsucc c'$. Given $S \subseteq \mathcal{C}$, $ic(S)$ denotes the set of strictly increasing sequences in $S^*$. More precisely, $ic(S) = \{c_1 \ldots c_n \in S^* \mid c_1 \prec c_2 \prec \ldots \prec c_n\}$. As before, we extend the underlying constraint system signature with constant predicates $is_b$ for each sequence $\beta \in \mathcal{C}^*$. These predicates are also decreed “private” in the sense that they are only allowed to occur in the process contexts $U_{oc}(S)[.]$ defined below.

**Definition 7.2.8 (Context $U_{oc}$).** The distinguishing context wrt $S \subseteq \mathcal{C}$, written $U_{oc}(S)$, is defined as

\[ ! ( \sum_{\beta \in ic(S)} \text{tell}(is_b) \parallel \mathcal{T}_\beta) \parallel [.] \]

where for each $\beta \in S^*$, $\mathcal{T}_{c,\beta} = \text{tell}(c) \parallel \mathcal{W}_\beta$ and $\mathcal{W}_{c,\beta} = \text{when } c \text{ do } \mathcal{T}_\beta$ with $\mathcal{T}_c = \mathcal{W}_c = \text{skip}$.

The role of the $is_b$'s predicates is similar to that of the $p_c$'s: to witness the $\mathcal{T}_\beta$ that the summation in $U_{oc}(S)$ chooses. The process $\mathcal{T}_\beta$ represents an increasing ask and tell interaction sequence. That is, if $\beta = c_1 \cdot c_2 \cdot c_3 \ldots$ then $\mathcal{T}_\beta$ first tells $c_1$ then asks $c_2$, then tells $c_3$ and so on. Intuitively, the role of the $\mathcal{T}_\beta$'s is to simulate the ask and tell sequences of interactions that a process can engage in.

The following theorem states that $U_{oc} \mathcal{C}$ is the universal context for the output congruence, and thus from Theorem 7.1.5, also for input-output congruence.

**Theorem 7.2.9 (Output Congruence Universal Context).** Suppose that $\mathcal{C}$ is finite. Then

\[ P \approx_o Q \iff U_{oc}(\mathcal{C})[P] \approx_o U_{oc}(\mathcal{C})[Q]. \]

**Proof.** The “only if” direction is trivial. Here we outline the proof of the “if” direction. From Lemma 7.2.3 it is sufficient to prove that $U_{oc}(\mathcal{C})[P] \approx_o U_{oc}(\mathcal{C})[Q]$ implies $R \parallel P \approx_o R \parallel Q$ for all $R$. Suppose that $R$ is such that $R \parallel P \not
approx_o R \parallel Q$. We want to prove that $U_{oc}(\mathcal{C})[P] \not
approx_o U_{oc}(\mathcal{C})[Q]$.

Consider the case $o(R \parallel P) \not
approx o(R \parallel Q)$. Take an $\alpha = d_0, d_1 \ldots$ such that $\alpha \in o(R \parallel P)$ and $\alpha \not
approx o(R \parallel Q)$. Furthermore, suppose that $R_0 \parallel P_0 \Rightarrow R_1 \parallel P_1 \Rightarrow \ldots$ with $P = P_0$ and $R = R_0$. 

We can represent the internal reduction of each $R_i \parallel P_i$ which gives us $d_i$ and $R_{i+1} \parallel P_{i+1}$ as a sequence of internal transitions (or interactions)

$$
\langle R^0_i \parallel P^0_i, c^0_i \rangle \rightarrow^* \langle R^n_i \parallel P^n_i, c^n_i \rangle \rightarrow^0 \langle R^n_i, d_i, c^n_i \rangle, \text{ with } R_i = R^0_i, P_i = P^0_i, c_i = \text{true}, P_{i+1} = F(P^n_i), R_{i+1} = F(R^n_i) \text{ and } d_i = c^n_i, \text{ satisfying}
$$

$$
\langle P^0_i, a^0_i \rangle \rightarrow^* \langle P^1_i, a^1_i \rangle \rightarrow^* \langle P^2_i, a^2_i \rangle \rightarrow^* \langle P^3_i, a^3_i \rangle \rightarrow^* \langle P^{j+1}_i, a^{j+1}_i \rangle
$$

$$
\langle R^0_i, b^0_i \rangle \rightarrow^* \langle R^1_i, b^1_i \rangle \rightarrow^* \langle R^2_i, b^2_i \rangle \rightarrow^* \langle R^{j+1}_i, b^{j+1}_i \rangle
$$

where for each $j < n$, $c^j_i = a^j_i \land b^j_i$. Let $\sigma_i = b^1_i.c^1_i \ldots b^n_i.c^n_i$. It is easy to see that $\langle \mathcal{T}_{\sigma_i} \parallel P^0_i, c^0_i \rangle \rightarrow^* \langle \mathcal{T}_e \parallel P^n_i, c^n_i \rangle$ (see Definition 7.2.8). Note that sequence $\sigma_i$ is increasing, thus by removing all constraint repetitions we get a strictly increasing sequence. Let $\beta_i$ be such a sequence. One can verify that $\mathcal{T}_{\beta_i}$ can “mimic” $R^0_i$ interacting with $P^0_i$. More precisely, $\langle \mathcal{T}_{\beta_i} \parallel P^0_i, c^0_i \rangle \rightarrow^* \langle \mathcal{T}_e \parallel P^n_i, c^n_i \rangle$ for $\langle \mathcal{T}_{\beta_i} \parallel P^0_i, c^0_i \rangle$. This implies:

$$
\langle \left( \sum_{\beta \in c(C)} \text{tell}(is\beta) \parallel \mathcal{T}_{\beta} \parallel P^0_i, \text{true} \right) \rightarrow^* \langle \mathcal{T}_e \parallel P^n_i, d_i \land is\beta \rangle \rightarrow^0 \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·
7.2.2 The General Case

It is interesting that even if $\mathcal{C}$ is not finite, we can still construct specialized distinguishing contexts to establish results similar to those of the previous section.

The key idea is to choose for a given finite set of processes, say $\Omega$, a suitable finite set of relevant constraints out of the constraints appearing in $\Omega$.

**Relevant Constraints**

Let us give some intuition about the relevant constraints of a process.

**Example 7.2.10.** Let

$$P = \text{tell}(0 < x \land x < y) \parallel \text{when } \text{prime}(x) \text{ do } \text{tell}(z = 0)$$

where $\text{prime}(x)$ holds iff $x$ is a prime number. Note that $P$ tells $(0 < x \land x < y)$ no matter what, and on input $\text{prime}(x)$, which occurs as a constraint in $P$, tells $z = 0$. But $P$ can also tell $z = 0$ on other inputs, e.g., $y = 4$ or $x = 7$ none of which appears in $P$. Finally, there are inputs such as $x = 42$ on which $P$ cannot tell $z = 0$.

Consider an arbitrary constraint $c$. $P$ on input $c$ tells $z = 0$ iff the assertion $(c \land 0 < x \land x < y) \models \text{prime}(x)$ holds. But such an assertion holds if and only if $c \models (0 < x \land x < y) \Rightarrow \text{prime}(x)$. Let us declare $c' = ((0 < x \land x < y) \Rightarrow \text{prime}(x))$. We notice that $P$ on $c'$ can also tell $z = 0$. In fact, one can verify that

$$P \xrightarrow{(c, \land \Delta)} P' \text{ iff } P \xrightarrow{(c', \land \Delta)} P'.$$

Try for example $c = (x = 42)$ or $c = (y < 4)$. The relevance of $c'$ is that characterizes $P$ on every input $c$ and it can be obtained from the constraints in $P$ using implication.

Now let $Q = (\text{local} \ x) P$. Notice, that like $P$, $Q$ tells $z = 0$ on inputs such as $y = 4$. But because $x$ is a local variable, unlike $P$, $Q$ does not tell $z = 0$ on inputs such as $\text{prime}(x)$. For the same reason, $Q$ does not tell $z = 0$ on $c'$ above.

Let us now define $e = \forall_x c'$. Notice that $Q$ on input $e$, tells $z = 0$. Actually, one can verify that

$$Q \xrightarrow{(e, \land \Delta)} Q' \text{ iff } Q \xrightarrow{(e, \land \Delta)} Q'. $$

Notice that $e$ can be obtained from the constraints in $Q$ using implication and universal quantification.

Let $R$ the process that results from replacing in $Q$, $\text{tell}(0 < x \land x < y)$ with $\text{tell}(0 < x) \parallel \text{tell}(x < y)$. The process $R$ is obviously input-output equivalent to $Q$. Therefore,

$$R \xrightarrow{(e, \land \Delta)} R' \text{ iff } R \xrightarrow{(e, \land \Delta)} R'. $$

In this case, however, $e = \forall_x c'$ can be obtained from the constraints in $R$ using implication, conjunction and universal quantification.

Finally, notice that $(\exists z < x)$ is equivalent to $(0 < x)$ if the underlying constraint system is, for example, modular arithmetic (Definition 3.1.5). So, in
modular arithmetic $c'$ above is equivalent to $((\exists z < x) \land x < y) \Rightarrow z = 0$. Thus, we could have used $\text{tell}(\exists z < x)$ instead of $\text{tell}(0 < x)$. But $\text{tell}(\exists z < x)$ is input-output congruent to $(\text{local} z) \text{tell}(z < x)$. Therefore, let $S$ be the process that results from replacing in $R$ above $\text{tell}(0 < x)$ with $(\text{local} z) \text{tell}(z < x)$.

One can verify that

$$S \xrightarrow{(c, c \land d)} S' \text{ iff } S \xrightarrow{(c, c \land d)} S'. \tag{7.2.10}$$

In this case $c = \forall x c'$ can be obtained from the constraints in $S$ using implication, conjunction, universal and existential quantification.

The above examples illustrates how to obtain relevant inputs for processes out of the constraint appearing in them by using implication, conjunction, and if local operators are involved, also universal and existential quantification. Next we shall show that it is all what we need.

**Definition 7.2.11 (Relevant Constraints).** Given $S \subseteq C$, let $\overline{S}$ be the closure under conjunction and implication of $S$. Let $C : \text{Proc} \to \mathcal{P}(C)$ be defined as:

- $C(\text{skip}) = \{ \text{true} \}$
- $C(\text{tell}(c)) = \{ c \}$
- $C(\sum_{i \in I} \text{when } c_i \text{ do } P_i) = \bigcup_{i \in I} \{ c_i \} \cup C(P_i)$
- $C(\text{unless } c \text{ next } P) = \{ c \} \cup C(P)$
- $C(P || Q) = C(P) \cup C(Q)$
- $C(\text{l P}) = C(\ast P) = C(\text{next } P) = C(P)$
- $C((\text{local} x) P) = \{ c, \exists z c, \forall z c \mid c \in C(P) \}$

Let $\Omega = \{ P_1, \ldots, P_n \}$. Define the relevant input constraints for processes in $\Omega$, written $C(\Omega)$, as the closure under conjunction of the set $C(P_1) \cup \ldots \cup C(P_n)$.

**Definition 7.2.12 (Strongest Consequence).** Assume that $P \in \Omega \subseteq \text{fin Proc}$. Define the strongest consequence of $d$ in $\Omega$, written $d(\Omega)$, as the unique constraint (modulo logical equivalence) $e \in C(\Omega)$ such that $d \models e$ and $e \models e'$ for every $e' \in C(\Omega)$ such that $d \models e'$.

Notice that that $d(\Omega)$ always exists since $C(\Omega)$ is closed under conjunction. Furthermore, it can be computed since $\models$ is decidable and $C(\Omega)$ is finite.

The next lemma is the key of the results in this section. Intuitively it states that $C(\Omega)$ indeed contains the relevant input constraints of $P \in \Omega$.

**Lemma 7.2.13.** Assume that $P \in \Omega \subseteq \text{fin Proc}$. Then

$$P \xrightarrow{(c, c \land d)} P' \text{ iff } P \xrightarrow{(c(\Omega), c(\Omega) \land d)} P'. \tag{7.2.11}$$

**Proof.** It suffices to show that

$$(P, c) \rightarrow^* (Q, c \land d) \text{ iff } (P, c(\Omega)) \rightarrow^* (Q, c(\Omega) \land d)$$

Consider the “only if” direction. To simplify the presentation of this proof, let us assume that $P$ contains no nesting of local operator. Each sequence
7.2. Characterization of Behavioral Equivalences

\( \langle P_0, c \rangle \rightarrow^* \langle P_n, c \land d \rangle \), with \( P_0 = P \) and \( P_n = Q \), can be represented as a sequence:

\[
\langle P_0, c \rangle \rightarrow^* \langle P_1, c \land c_1 \rangle \rightarrow \langle P'_1, c \land c_1 \rangle \rightarrow^* \ldots \rightarrow^* \langle P_i, c \land c_i \rangle \rightarrow \langle P'_{i+1}, c \land c_{i+1} \rangle \rightarrow^* \ldots
\]

satisfying the conditions below. The (zero or more) reductions \( \langle P_i, c \land c_i \rangle \rightarrow \langle P'_{i+1}, c \land c_{i+1} \rangle \) are obtained from a derivation whose topmost (or root) rule is either SUM or UNL. In other words the reduction involves the execution of either a summation or an unless operator. Furthermore, each of the \( \langle P_i, c \land c_i \rangle \rightarrow^* \langle P_{i+1}, c \land c_{i+1} \rangle \) involves no application of SUM or UNL.

Suppose that \( g_i \) is the constraint guard of the summation or unless operator when deriving \( \langle P_i, c \land c_i \rangle \rightarrow \langle P'_{i+1}, c \land c_{i+1} \rangle \). We can infer that

\[
e_i \land \exists x_i (c \land c_i) \models g_i
\]

where \( x_i \) is vector of at most one variable and \( e_i \) is a local information introduced by rule LOC (the vector can be empty and \( e_i \) can be true meaning that LOC was not applied). Notice that

\[
e_i \land \exists x_i (c \land c_i) \models g_i \iff \exists x_i (c \land c_i) \models (e_i \Rightarrow g_i) \\
e_i \land \exists x_i (c \land c_i) \models \forall x_i (e_i \Rightarrow g_i) \\
e_i \land c \land c_i \models \forall x_i (e_i \Rightarrow g_i) \\
e_i \land c \land c_i \models \Rightarrow x_i (e_i \Rightarrow g_i)
\]  

(7.5)

Let \( d_i = c_i \Rightarrow \forall x_i (e_i \Rightarrow g_i) \). From Definition 7.2.11, \( g_i \in C(\Omega) \) since \( g_i \) appears as a guard in \( P \). One can verify that \( c_i \) can be constructed out of the constraints in tell operators via conjunction and existential quantification. Hence \( c_i \in C(\Omega) \) by Definition 7.2.11. Similarly, since \( e_i \) represents local information, one can verify that it can be constructed out of the constraints in tell operators some local process in \( P \) via conjunction and existential quantification, hence \( e_i \in C(\Omega) \). Therefore, from Definition 7.2.11, \( d_i \in C(\Omega) \).

Let \( d' = \bigwedge_{i \in \{1, \ldots, n\}} d_i \). By induction on \( n \) we can show that \( \langle P_0, c' \rangle \rightarrow^* \langle P_n, c' \land d \rangle \). From the equivalences in Equation 7.5 \( c \models d' \). Moreover, \( c' \in C(\Omega) \) since each \( d_i \in C(\Omega) \) and \( C(\Omega) \) is closed under conjunction. Hence \( c \models c(\Omega) \models d' \) by Definition 7.2.12. We can then verify that \( \langle P_0, c(\Omega) \rangle \rightarrow^* \langle P_n, c(\Omega) \land d \rangle \) as wanted.

The "if" direction can be obtained in a similar way by reversing the arguments of the above case and the equivalences in Equation 7.5.

**Specialized Distinguishing Contexts**

Having shown that the finite set of constraints \( C(\Omega) \) contains the relevant input for \( \Omega \) processes, we now proceed to give characterizations of the various equivalence similar those given in Section 7.2.1.

The following proposition tells us that to determine whether \( P \) and \( Q \) have different postcondition we verify whether \( P \) and \( Q \) can be distinguished via output equivalence by the context \( U_{ap}(C(\{P, Q\})) \) (Definitions 7.2.4 and 7.2.11).
Theorem 7.2.14 (SP Distinguishing Context). Assume that $P, Q \in \Omega \subset \text{fin Proc.}$ Then

$$P \sim_{sp} Q \text{ iff } U_{sp}(\mathcal{C}(\Omega))[P] \sim_{o} U_{sp}(\mathcal{C}(\Omega))[Q].$$

Proof. The proof proceeds very much as the proof of Proposition 7.2.5, except for in this case we appeal to Lemma 7.2.13. Here we illustrate the “if” direction. Suppose that $U_{sp}(\mathcal{C}(\Omega))[P] \sim_{o} U_{sp}(\mathcal{C}(\Omega))[Q].$ Consider the case $sp(P) \subseteq sp(Q).$ Suppose that $\alpha = c_1, c_2, \ldots \in sp(P).$ We want to prove $\alpha \in sp(Q)$.

Let $R = \sum_{c \in \mathcal{C}(\Omega)} \text{tell}(c).$ Since $\alpha \in sp(P)$, then from Theorem 3.5.2 it follows that

$$P = P_1 \xrightarrow{(c_1, c_1)} P_2 \xrightarrow{(c_2, c_2)} \ldots \quad (7.6)$$

Notice that

$$R \xrightarrow{\text{true}, c_1(\Omega)} R \xrightarrow{\text{true}, c_2(\Omega)} \ldots \quad (7.7)$$

Then by Lemma 7.2.13 applied to Equation 7.6, and Equation 7.7 we can verify that

$$R \parallel P_1 \xrightarrow{\text{true}, c_1(\Omega)} R \parallel P_2 \xrightarrow{\text{true}, c_2(\Omega)} \ldots$$

Therefore $\alpha' = c_1(\Omega), c_2(\Omega) \ldots \in o(R \parallel P)$ and then from the initial assumption $\alpha' \in o(R \parallel Q)$. Hence, $\alpha' \in sp(R \parallel Q)$ by Theorem 3.5.2. From this we can prove that $\alpha' \in sp(Q)$. Then it follows, with the help of Lemma 7.2.13, that $\alpha \in sp(Q)$ as wanted.

The case $sp(Q) \subseteq sp(P)$ is symmetric. $\Box$

Similarly, we have the following result reducing input-output equivalence of $P$ and $Q$ to output equivalence of $P$ and $Q$ in a single context.

Theorem 7.2.15 (IO Distinguishing Context). Assume that $P, Q \in \Omega \subset \text{fin Proc.}$ Then

$$P \sim_{io} Q \text{ iff } U_{io}(\mathcal{C}(\Omega))[P] \sim_{o} U_{io}(\mathcal{C}(\Omega))[Q].$$

Proof. Similar to the proof of Proposition 7.2.7. Here we illustrate the “if” direction. Suppose that $U_{io}(\mathcal{C}(\Omega))[P] \sim_{o} U_{io}(\mathcal{C}(\Omega))[Q]$. Consider the case $io(P) \subseteq io(Q)$. Let $\alpha = c_1, c_2, \ldots$ and $\alpha' = c'_1, c'_2, \ldots$ such that $(\alpha, \alpha') \in io(P)$. We want to prove $(\alpha, \alpha') \in io(Q)$. Let $d_1, d_2, \ldots$ be the constraints such that $\alpha' = (c_1 \land d_1), (c_2 \land d_2), \ldots$.

Let $R = \sum_{c \in \mathcal{C}(\Omega)} \text{tell}(c) \parallel \text{tell}(p_c)$. Notice that

$$R \xrightarrow{\text{true}, c_1(\Omega) \land p_{c_1}(\Omega)} \xrightarrow{\text{true}, c_2(\Omega) \land p_{c_2}(\Omega)} \ldots$$

With the help of Lemma 7.2.13 we can show that $(c_1(\Omega), c_2(\Omega), \ldots, (c_1(\Omega) \land d_1), (c_2(\Omega) \land d_2), \ldots) \in io(P)$. It should then be the case that, for $P = P_1$, $P = P_1 \xrightarrow{(c_1, c_1 \land p_{c_1}(\Omega))} P_2 \xrightarrow{(c_2, c_2 \land p_{c_2}(\Omega))} \ldots$
From the initial assumption it follows that for \( Q = Q_1 \),
\[
R \parallel Q_1 \overset{(\text{true}, c_1(\Omega) \land d_1 \land p_{c_1})}{\longrightarrow} R \parallel Q_2 \overset{(\text{true}, c_2(\Omega) \land d_2 \land p_{c_2})}{\longrightarrow} \ldots
\]

From our assumption about the privacy of the \( p_c \)'s predicates, we conclude that, since \( p_{c_1}(\Omega) \) is told at time \( i \), \( R \) chose \text{tell}(c_i(\Omega)) \) at time \( i \). From this we can then conclude
\[
Q_1 \overset{(c_1(\Omega) \land p_{c_1}(\Omega), c_1(\Omega) \land d_1 \land p_{c_1}(\Omega))}{\longrightarrow} Q_2 \overset{(c_2(\Omega) \land p_{c_2}(\Omega), c_2(\Omega) \land d_2 \land p_{c_2}(\Omega))}{\longrightarrow} \ldots
\]

Again from the privacy of the \( p_c \)'s predicates it follows that
\[
Q_1 \overset{(c_1(\Omega), c_1(\Omega) \land d_1)}{\longrightarrow} Q_2 \overset{(c_2(\Omega), c_2(\Omega) \land d_1)}{\longrightarrow} \ldots
\]

It then follows from Lemma 7.2.13 that \( (\alpha, \alpha') \in \text{io}(Q) \) as wanted. The case \( \text{io}(Q) \subseteq \text{io}(P) \) is symmetric. \( \square \)

Finally, the next result also reduces the problem of determining whether \( P \) and \( Q \) are output-congruent (and therefore input-output congruent) to whether they are output equivalent in one specific context.

**Theorem 7.2.16 (Output Congruence Distinguishing Context).** Assume that \( P, Q \in \Omega \subset_{\text{fin}} \text{Proc} \). Then
\[
P \approx_o Q \iff U_{\alpha_c}(C(\Omega))[P] \sim_o U_{\alpha_c}(C(\Omega))[Q].
\]

**Proof.** The proof is the same as that of Theorem 7.2.9 except for the role of \( \beta_i \) which is now played by the sequence \( \beta_i \) which results from replacing each \( e \) in \( \beta_i \) with \( e(\Omega) \).

As in the proof of Lemma 7.2.13 we can prove that a constraint in \( C(\Omega) \) can be inferred from \( e \) iff it can be inferred from \( e(\Omega) \). We then proceed exactly as in the proof of Theorem 7.2.9 until Equations (7.3) and (7.4), which we re-state as:
\[
\langle \bigl( \sum_{\beta \in C(\Omega)} \text{tell}(is_{\beta}) \parallel \mathcal{T}_\beta \parallel P_1^0, \text{true} \bigr) \overset{\rightarrow}{\longrightarrow}^* \langle \mathcal{T}_e \parallel P_1^0, d_i(\Omega) \land is_{\beta_i} \rangle \rangle \rightarrow
\]

and

for all \( P' \in \Omega,
\[
\text{if } \langle \mathcal{T}_\beta \parallel P', \text{true} \rangle \overset{\rightarrow}{\longrightarrow}^* \langle \mathcal{T}_e \parallel P', d_i(\Omega) \rangle \rightarrow \text{, with } P' \overset{\rightarrow}{\longrightarrow}^* P'',
\]
then \( \langle R_1^0 \parallel P', \text{true} \rangle \overset{\rightarrow}{\longrightarrow}^* \langle R_1^0 \parallel P'', d_i \rangle \rightarrow \) \( (7.9) \)

We then proceed as in the proof of Theorem 7.2.9; getting a contradiction out of (7.8) and (7.9). \( \square \)
The above theorem allows us to think of $U_{oc}(C(\Omega))$ as the universal context for the processes in $\Omega$. The ability of constructing distinguishing contexts for arbitrary processes is important since, as we can infer from the above results, it can be used for proving decidability results for $\approx_{\varphi}$, $\approx_{\sigma}$, $\approx_{\varphi}$, $\approx_{\varphi}$ by a reduction to $\approx_{\sigma}$. It turns out that $\approx_{\sigma}$ is decidable for a significant fragment of the calculus. The languages of these processes can be recognized by automata over infinite sequences, more precisely Büchi Automata [Buc62]. We will elaborate on this in the next section.

### 7.3 Decidability of Observable Equivalences

We aim at proving decidability of the relation $\approx_{\sigma}$ for the fragment of ntcc which we have called restricted-nondeterministic processes. These are basically star-free processes in which nondeterminism in local operator is not allowed. We then show, by using the characterizations given in the previous sections of the various equivalences, that the input-output and strongest-postcondition equivalences as well as output and input-output congruences are also decidable for such a fragment.

Let us now define formally what we mean by restricted-nondeterministic processes. Recall that a choice operator $\sum_{i \in I} \text{when } c_i \text{ do } P_i$ is mutually exclusive iff all the $c_i$'s are pairwise mutually exclusive.

**Definition 7.3.1.** A star-free process $P$ is restricted-nondeterministic iff given an arbitrary (local $x$) $Q$ in $P$, every choice operator in $Q$ is mutually exclusive. We use $\text{Proc}^r$ to denote the set of all restricted-nondeterministic processes.

This fragment allows non-deterministic process (summations) out of the scope of local variables. In fact, all application examples in Chapter 6 belong to this fragment. Notice that each local $x$ in $P \in \text{Proc}^r$ is deterministic in the sense of Definition 4.4.1.

In order to prove the decidability result for $\approx_{\sigma}$, we shall characterize the output behavior of restricted-nondeterministic processes in terms of $\omega$-regular languages, i.e., the languages accepted by Büchi automata [Buc62]. Recall that Büchi automata are ordinary nondeterministic finite-state automata equipped with an acceptance condition that is appropriate for $\omega$-sequences: an $\omega$-sequence is accepted if the automaton can read it from left to right while visiting a sequence of states in which some final state occurs infinitely often. We shall then use the fact that language equivalence for Büchi automata is decidable [SVW87], to conclude that $\approx_{\sigma}$ is decidable for restricted-nondeterministic processes.

#### 7.3.1 Infinite Number of States from Nondeterminism

Let us illustrate the problem in trying to use finite-state machines for representing arbitrary processes. Recall that $P \xrightarrow{d} Q$ holds iff $P \xrightarrow{\text{true}, d} Q$.

**Example 7.3.2.** Let $Q = \text{!!}P$ with $P = \sum_{j \in J} \text{tell}(c_j)$. We have the following transition sequence (on input $\text{true}^\omega$):

$$Q \xrightarrow{d_1} Q \xrightarrow{d_2} Q \xrightarrow{d_3} Q \xrightarrow{d_4} \cdots \xrightarrow{d_n} Q \xrightarrow{\prod_{1}^{n} \text{!!}P} \xrightarrow{d_{n+1}} \cdots$$
This example illustrates that in a transition system where states are the elements of $Proc$ it is possible to have infinite paths where all states are different up to the structural congruence $\equiv$. In other words, there can be an infinite set of derivatives. Moreover, notice that in the above example, the process at time $i$ can output everything the process at time $i - 1$ can, but not necessarily the other way round. This situation arises from the nondeterminism specified by $P$ and the replication operator.

Nevertheless, we wish to show that after some time units the states can be identified up to $\approx_o$. More precisely, the property we wish to have is:

$$\text{There exists } t \text{ s.t. for all } k \geq t, \prod_{k} P \approx_o \prod_{k+1} P.$$  \hspace{1cm} (7.10)

In the above example for any $k \geq |J|$ we have $\prod_k P \approx_o \prod_{k+1} P$ thus validating the property.

Unfortunately, the property does not hold in general, as the following counterexample shows.

**Example 7.3.3.** Let us define an arbitrary-delay operation $\delta P$ which delays $P$ arbitrarily:

$$\delta P \overset{\text{def}}{=} P + \delta P.$$  

The encoding in our calculus of the recursive definition of $\delta P$ requires hiding over (non-mutually exclusive) summations (see Section 6.1) thus it is out of $Proc^\tau$. Assume that $P = \text{tell}(c)$. Then two copies of $\delta P$ can output $c$ at two (arbitrary) points of time while a single copy cannot. In general one can prove that for any $k > 1$, $o(\prod_k \delta P) \subset o(\prod_{k+1} \delta P)$, thus invalidating Property 7.10. The same situation occurs with process $*P$ which also specifies nondeterminism.

We shall show in the next section that Property 7.10 indeed holds for processes in $Proc^\tau$.

### 7.3.2 Finite-State Representation for Restricted Nondeterminism

In the previous section we saw that there cannot be a finite-state representation of the output behavior of processes (even in the restricted-nondeterministic fragment) if the states are just processes modulo the congruence $\equiv$. In this section we shall introduce a decidable congruence $\equiv_c$, weaker than $\equiv$ but stronger than the output congruence $\approx_o$. Such a congruence will allow for the finite-state representation of output behavior by taking the states processes modulo $\equiv_c$ rather than $\equiv$.

The following proposition is needed in the proof of Lemma 7.3.6 which implies Property 7.10. It relates the output behavior of processes with that of processes arising at intermediate steps of the internal computations.

**Proposition 7.3.4.** $\alpha \in o(P)$ iff there are $Q$ and $c$ such that $\langle P, \text{true} \rangle \xrightarrow{\alpha}^* \langle Q, c \rangle$ and $Q \parallel \text{tell}(c) \xrightarrow{\alpha} \omega.$
We also need the notion of multiplicity of a process:

**Definition 7.3.5.** Let \( m : \text{Proc}' \to \mathcal{N} \). The multiplicity of \( P \), \( m(P) \) is defined as

\[
\begin{align*}
m(\text{skip}) &= 0 \\
m(\text{tell}(c)) &= 1 \\
m(\sum_{i \in I} \text{when } c_i \text{ do } P_i) &= \sum_{i \in I} m(P_i) \\
m(P \parallel Q) &= \max\{m(P), m(Q)\} \\
m(\text{local } x \text{ in } P) &= m(\text{next } P) = m(\text{unless } c \text{ next } P) = m(P) = m(P).
\end{align*}
\]

The multiplicity of \( P \), \( m(P) \), is aimed to be the number of copies of \( P \), after which, further copies are redundant. This is stated in the following lemma which is the key for decidability of \( \sim_o \).

**Lemma 7.3.6.** Let \( P \in \text{Proc}' \). For all \( k > m(P) \), \( \Pi_{k-1} P \equiv_o \Pi_k P \).

**Proof.** The proof proceeds by induction on the structure of \( P \in \text{Proc}' \). Suppose \( k > m(P) \).

\( P = \text{tell}(c) \). Immediate.

\( P = P_1 \parallel P_2 \). Since \( \equiv \subseteq \approx_o \) by Theorem 7.1.5 and \( \Pi_k (P_1 \parallel P_2) \equiv \Pi_k P_1 \parallel \Pi_k P_2 \) (see Definition 3.3.4), we get \( \Pi_k (P_1 \parallel P_2) \approx_o \Pi_k P_1 \parallel \Pi_k P_2 \). Note that \( k > m(P) \geq m(P_1) \) and \( k > m(P) \geq m(P_2) \). Therefore, from the hypothesis \( \Pi_k P_1 \parallel \Pi_k P_2 \approx_o \Pi_{k-1} P_1 \parallel \Pi_{k-1} P_2 \approx_o \Pi_{k-1} (P_1 \parallel P_2) \) as required.

\( P = \text{next } Q \). It easy to verify that \( \Pi_n \text{next } Q \approx_o \Pi_n Q \) for every \( n \). Furthermore, from the hypothesis it follows that \( \Pi_k Q \approx_o \Pi_{k-1} Q \). Thus, \( \Pi_k \text{next } Q \approx_o \Pi_k Q \approx_o \Pi_{k-1} \text{next } Q \). Thus, \( \Pi_k \text{next } Q \approx_o \Pi_k Q \approx_o \Pi_{k-1} \text{next } Q \).

\( P = \text{unless } c \text{ next } Q \). Similar to the previous case.

\( P = !Q \). We can verify that \( \Pi_n !Q \approx_o !\Pi_n Q \) for all \( n \). From the hypothesis, \( \Pi_k Q \approx_o \Pi_{k-1} Q \). Then \( \Pi_k !Q \approx_o !\Pi_k Q \approx_o !\Pi_{k-1} Q \).

\( P = \sum_{u \in I} \text{when } c_u \text{ do } P_u \). From Lemma 7.2.3 we know that it is sufficient to consider parallel contexts. Let \( E \) an arbitrary process and suppose that \( \alpha = c, \alpha' \in o(\mathcal{E} \parallel \Pi_k P) \) (1). We want to show that \( \alpha \in o(E \parallel \Pi_k P) \). From (1) we know that there exists sequence of internal transitions \( t = \langle E \parallel \Pi_k P, \text{true} \rangle \rightarrow^* \gamma_1 \rightarrow^* \ldots \rightarrow^* \gamma_n \rightarrow^* \langle R, c \rangle \rightarrow^* \) with \( \alpha' \in o(\mathcal{F}(R)) \) which contains only the initial and final configuration, and those configurations \( \gamma_1, \ldots, \gamma_n \) in which a reduction from a \( P \) takes place, if any. By monotonicity of the store if \( t \) contains a configuration with store \( c \) s.t. \( \langle P, c \rangle \rightarrow^* \) then since a reduction of each \( P \) must eventually take place \( n = k \) (I) otherwise \( n = 0 \) (II).

(I). Suppose \( n = k \). Define \( E_0 = E \), \( P_0 = \text{skip} \). For \( 0 < j \leq n \), each \( \gamma_j \) can be defined as \( \langle E_j \parallel P_j \parallel \Pi_{n-j} P, c_j \rangle \), where \( \langle E_{j-1} \parallel P_{j-1}, c_{j-1} \rangle \rightarrow^* \langle E_j, c_j \rangle \) for some \( c_j \) s.t. \( \langle P, c_j \rangle \rightarrow^* \langle P_j, c_j \rangle \) (a reduction from one of the
k P’s). From the assumption $k > m(P) = \sum_{Q \vdash_\Sigma P \rightarrow Q} m(Q)$, so from the pigeon-hole principle there must be a process $P’, P \rightarrow P’$ with $r > m(P’)$ configurations $\gamma_i, \ldots, \gamma_j$, such that each corresponding $P_{\gamma_i}, \ldots, P_{\gamma_j}$ is $P’$. Let $\gamma_i$ be the first among these configurations and let $P_i$ be the process in such a configuration, i.e., $E_i \parallel P’ \parallel \Pi_{k-i} P$. From Proposition 7.3.4, we have $\alpha \in o(P_i \parallel \text{tell}(c_i))$. As $r$ copies of $P’$ are eventually triggered, one can verify that $\alpha \in o(E_i \parallel \Pi_r P’ \parallel \Pi_{k-(i+r-1)} P \parallel \text{tell}(c_i))$. Since $P’$ is a subprocess of $P$, from the hypothesis $\alpha \in o(Q_i \parallel \text{tell}(c_i))$ with $Q_i = E_i \parallel \Pi_{r-1} P’ \parallel \Pi_{k-(i+r-1)} P$. One can then construct the sequence

$$
\langle E \parallel \Pi_{(k-1)} P, \text{true} \rangle \rightarrow^* \langle E_i \parallel P’ \parallel \Pi_{(k-1)-i} P, c_i \rangle \\
\rightarrow \langle E_i \parallel \Pi_2 P’ \parallel \Pi_{(k-1)-(i+1)} P, c_i \rangle \\
\vdots \\
\rightarrow \langle E_i \parallel \Pi_{r-1} P’ \parallel \Pi_{(k-1)-(i+r-2)} P, c_i \rangle \\
= \langle Q_i, c_i \rangle.
$$

From Proposition 7.3.4, $\alpha \in o(E \parallel \Pi_{(k-1)} P)$ as required.

(II) Suppose $n = 0$. Then $R = E’ \parallel \Pi_k P$ for some $E’$ such that $\langle E, \text{true} \rangle \rightarrow^* \langle E’, c \rangle \not\rightarrow$. Trivially $\langle E \parallel \Pi_{k-1} P, \text{true} \rangle \rightarrow\langle R’, c \rangle \not\rightarrow$ with $R’ = E’ \parallel \Pi_{k-1} P$. From the definition of $F(\_)$, $F(P) \equiv \text{skip}$, thus $F(R) = F(E’) \parallel \Pi_k F(P) \equiv F(E’) \equiv F(R’) = F(E’) \parallel \Pi_{k-1} F(P)$. We therefore have $F(R) \approx_o F(R’)$ since $\equiv \subset \approx_o$ by Theorem 7.1.5, and thus $\alpha \in F(R’).$ We then conclude $\alpha \in o(E \parallel \Pi_{k-1} P)$.

$P = (\text{local} x) Q$. In this case $P$ is a deterministic process. It is easy to verify that if $P$ is a deterministic process then $P \approx_o \Pi_k P$ for any $k$, thus validating the property.

$$
\square
$$

Intuitively, Lemma 7.3.6 above, tells us that if $R$ has $r$ copies of $P$ in parallel then we can freely “compact” $R$ by removing $r - m(P)$ of them without changing the output behavior. The equivalence relation $\equiv_c$ defined next, captures the above intuition by extending the structural congruence $\equiv$ with the axiom $\Pi_k P \equiv \Pi_{k-1} P$ if $k > m(P)$.

**Definition 7.3.7 (Compact Structural Congruence).** The relation $\equiv_c$ is the smallest congruence relation including $\equiv$ (Definition 3.3.4) and satisfying the axiom:

$$
\Pi_k P \equiv_c \Pi_{k-1} P \text{ if } k > m(P)
$$

**Proposition 7.3.8.** $\equiv \subset \equiv_c \subset \approx_o$

**Proof.** Immediate consequence of Definition 7.3.7 and Lemma 7.3.6. $
\square$

The modified standard form in the case of $\equiv_c$ arises as restriction of that of $\equiv$ (Definition 3.3.5) by compacting it according to Lemma 7.3.6. More precisely,
Definition 7.3.9. A process $P = \prod_{i \in I} E_i$ is in standard compact form if each $E_i$ appears, as component of the product, at most $m(E_i)$ times and takes one of the following forms

\[ \text{tell}(c), \sum_{j \in J} \text{when } c_j \text{ do } P_j, (\text{local } x) Q, \text{next } Q, \text{unless } c \text{ next } Q, * Q, ! Q, \]

with $|J| > 0$ and each $P_j$ and $Q$ being themselves in standard compact form. (If $I = \emptyset$ we take the standard compact form to be skip).

For example, $\text{tell}(c)$ is in both standard form and standard compact form. However, $\text{tell}(c) \parallel \text{tell}(c)$ is in standard form but it is not in standard compact form.

Proposition 7.3.10. Every process $P$ is structurally compact to a process in standard compact form.

Proof. The proof proceeds by induction on the structure of $P$. \qed

The decidability of $\equiv_c$ can be proven by using a graph representation of process terms in standard compact form and the above proposition.

Proposition 7.3.11. Relation $\equiv_c$ is decidable.

Proof. The proof proceeds as the proof of decidability of $\equiv$ given in Proposition 3.3.6: Using Proposition 7.3.10 and a reduction to graph isomorphism with the graph representation of processes (in this case, in standard compact form) given in such a proof. \qed

In Example 7.3.2 we showed that the set of derivatives modulo $\equiv$ of a given restricted-nondeterministic process can be infinite. The next lemma states that this situation changes if we consider $\equiv_c$ instead. Recall that $P \rightarrow Q$ holds iff $P \xrightarrow{(c,d)} Q$ for some $c$ and $d$.

Definition 7.3.12. A process $Q$ is said to be a derivative of $P$ iff there is a reduction sequence

\[ P = P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n = Q. \]

We define $D(P)$ as the set of all derivatives of $P$.

Lemma 7.3.13. The set $D(P)$ modulo $\equiv_c$ is finite.

Proof. The proof proceeds by induction on the structure of $P$. Let $n(P)$ be the number of derivatives of $P$ modulo $\equiv_c$, i.e., $|D(P)/ \equiv_c |$. We shall show that $n(P) < \omega$.

$P = \text{tell}(c)$. If $P = R_1 \rightarrow R_3 \rightarrow \ldots$ then for each $i > 0$ we can verify that $R_i \equiv \text{skip}$. Thus $n(P) = 1$.

$P = \sum_{i \in I} \text{when } c_i \text{ do } P_i$. If $P \rightarrow Q$ then either $Q \equiv \text{skip}$ or $Q$ is a derivative of $P_i$ for some $i \in I$. Hence, by appealing to induction we conclude that $n(P) \leq 1 + \sum_{i \in I} n(Q_i)$. 


7.3. Decidability of Observable Equivalences

\( P = P_1 \parallel P_2 \). If \( P = R_1 \implies R_2 \implies \ldots \), then for each \( i > 1 \) we can verify that \( R_i \equiv Q_i \parallel Q'_i \) where \( Q_i \) and \( Q'_i \) are derivatives of \( P_1 \) and \( P_2 \), respectively. We can then use the induction to conclude that \( n(P) \leq n(P_1) \times n(P_2) \).

\( P = (\text{local } x) Q \). If \( P = R_1 \implies R_2 \implies \ldots \) then for each \( i > 1 \) we can verify that \( R_i \equiv (\text{local } x) Q_i \) where \( Q_i \) is a derivative of \( Q \). Thus, by the induction \( n(P) \leq n(Q) \).

\( P = \text{next } Q \). If \( P \implies R \) then \( R \equiv Q \). Hence, by using induction we conclude that \( n(P) \leq 1 + n(Q) \).

\( P = \text{unless } c \text{ next } Q \). If \( P \implies R \) then either \( R \equiv \text{skip} \) or \( R \equiv Q \). Hence, by using induction we conclude that \( n(P) \leq 2 + n(Q) \).

\( P = \text{! } Q \). Notice that if \( \text{! } Q \implies R \) then \( R \equiv Q' \parallel \text{! } Q \) where \( Q \implies Q' \). Thus, by repeating this argument we can show that every sequence \( \text{! } Q \implies R_2 \implies \ldots \) is point-wise \( \equiv \) equivalent to a sequence

\[
\begin{align*}
\text{! } Q & \implies Q_{1,1} \parallel \text{! } Q \\
& \implies Q_{2,2} \parallel Q_{2,1} \parallel \text{! } Q \\
& \implies Q_{3,2} \parallel Q_{3,2} \parallel Q_{3,1} \parallel \text{! } Q \\
& \vdots 
\end{align*}
\]

(7.11)

where

\[
\begin{align*}
Q & \implies Q_{1,1} \implies Q_{2,2} \implies Q_{3,3} \implies \ldots \\
Q & \implies Q_{2,1} \implies Q_{3,2} \implies Q_{4,3} \implies \ldots \\
Q & \implies Q_{3,1} \implies Q_{4,2} \implies Q_{5,3} \implies \ldots \\
& \vdots 
\end{align*}
\]

It follows that each \( Q_{i,j} \) in Equation 7.11 is a derivative of \( Q \). We therefore conclude, by appeal to induction, that there must be finitely many \( Q_{i,j} \)'s.

Let \( R_i = \prod_{j \in \{1, \ldots, i\}} Q_{i,j} \), i.e., the product of the \( Q_{i,j} \)'s processes at time \( i \) in Equation 7.11.

Notice that to prove that \( n(\text{! } Q) < \omega \), it suffices to show that there are finitely many \( R_i \)'s modulo \( \equiv_c \). This follows from a basic combinatorial argument: We can construct only finitely many different products of processes in a finite set \( S \) if we restrict such products to have at most \( m(P') \) occurrences of each \( P' \in S \). In fact, \( 2^{|S| \times M} \), where \( M = \max \{n \mid n = m(P') \land P' \in S\} \) is an upper bound in the number of such different products. One can verify that \( m(\text{! } Q) \geq m(Q_{i,j}) \) for each of the \( Q_{i,j} \)'s. Therefore, from the above arguments, \( n(\text{! } Q) \leq 2^{m(\text{! } Q) \times m(\text{! } Q)} \).

\[ \square \]

7.3.3 Büchi Automata Representation

In this section we shall give Büchi automata representation for both the output and the input-output behavior of restricted-nondeterministic processes.
7.3.4 Output Büchi Automata

Given a restricted-nondeterministic process $P$ we can construct a Büchi automaton $A_o(P)$ which accepts $o(P)$. The start state is $P$ and each transition from state $Q$ to state $R$ with label $d$ represents an observable reduction $Q \xrightarrow{\text{true},d} R$. The construction is given by Algorithm 7.1 whose correctness is stated in Lemma 7.3.14 below.

Algorithm 7.1 Construction of Büchi Automaton $A_o(P)$

1. Make $P$ to be an accepting and the start state.

2. Choose a state $Q$ from the current transition graph and compute a reduction $Q \xrightarrow{\text{true},d} R$ (such computation always terminates). The choice should satisfy that there is not already an edge labeled with $d$ from $Q$ to some $R' \equiv_e R$. If such a choice is not possible then return the current automaton construction.

3. Else if there is already a state $R' \equiv_e R$ then create an edge labeled with $d$ from $Q$ to it. Otherwise, create a new (accepting) state $R$ and edge from $Q$ to it with label $d$.

4. Go to 2.

Lemma 7.3.14. Given $P \in \text{Proc}^r$, Algorithm 7.1 effectively constructs a Büchi automaton $A_o(P)$ which accepts the language $o(P)$.

Proof. The termination of Algorithm 7.1 follows from the decidability of $\equiv_e$ (Proposition 7.3.11) and the finiteness of the set of derivative modulo $\equiv_e$ of $P$ (Lemma 7.3.13). The partial correctness of Algorithm 7.1 can be easily verified by using Proposition 7.3.8.

Therefore, for restricted-nondeterministic processes, the question of whether $P$ and $Q$ have the same output behavior can be reduced to whether $A_o(P)$ and $A_o(Q)$ accept the same language. Since language equivalence for Büchi automata is decidable [SVW87], we can conclude that $\sim_o$ is decidable as well. Such a decidability result is stated in the next theorem. We wish to explicitly mention that the decidability depends on no assumptions about the underlying constraint system (other than the decidability of $|=\equiv_e$, of course).

Theorem 7.3.15. Relation $\sim_o$, restricted to processes in $\text{Proc}^r$ over arbitrary constraint systems, is decidable.

Proof. Immediate consequence of Lemma 7.3.14.

Recall that the characterizations given by Theorems 7.2.14, 7.2.15 and 7.2.16 reduce the various equivalences between two arbitrary processes to their output equivalence in a particular process context. Furthermore, the input-output and output congruence coincide for processes in $\text{Proc}^r$ (Theorem 7.1.5). Therefore,
Corollary 7.3.16. The relations \( \approx_o, =_{io}, \approx_{sp} \) and \( \approx_{sp} \), restricted to processes in \( \text{Proc}^r \) over arbitrary constraint systems, are decidable.

Proof. It follows immediately from Theorem 7.3.15 and Theorems 7.2.14, 7.2.15, 7.2.16, 7.1.5. Notice that the contexts in Theorems 7.2.14, 7.2.15 and 7.2.16 can be effectively constructed. \( \square \)

7.3.5 Input-Output Büchi Automata Representation

In this section we give a modified version of Algorithm 7.1 to construct an Büchi automaton \( A_{io}(P) \) for the relevant input-output behavior of \( P \). The start state is \( P \) and each transition from a state \( Q \) to a state \( R \) with label \( (c,d) \) represents an observable reduction \( Q \xrightarrow{(c,d)} R \) with \( c \) is in the finite set of relevant constraints for \( P \), \( C(\{P\}) \) (see Definition 7.2.11).

Algorithm 7.2 Construction of Büchi Automaton \( A_{io}(P) \)

1. Make \( P \) to be an accepting and the start state.

2. Choose a state \( Q \) from the current transition graph and compute a reduction \( Q \xrightarrow{(c,d)} R \) (such computation always terminates) with \( c \in C(\{P\}) \). The choice should satisfy that there is not already an edge labeled with \( (c,d) \) from \( Q \) to some \( R' \equiv_c R \). If such a choice is not possible then return the current automaton construction.

3. Else if there is already a state \( R' \equiv_c R \) then create an edge labeled with \( (c,d) \) from \( Q \) to it. Otherwise, create a new (accepting) state \( R \) and edge from \( Q \) to it with label \( (c,d) \).

4. Go to 2.

The correspondence between the input-output behavior of a given process \( P \) and the automaton \( A_{io}(P) \) constructed by Algorithm 7.2 is stated in the next lemma. Recall that \( c(\Omega) \) denotes the strongest constraint in \( C(\Omega) \) entailed by \( c \) (see Definition 7.2.12).

Lemma 7.3.17. Given \( P \in \text{Proc}^r \), Algorithm 7.2 effectively constructs a Büchi automaton \( A_{io}(P) \), s.t., \( (c_1,e_2,\ldots, (c_1 \land d_1),(c_2 \land d_2)\ldots) \in \text{io}(P) \) if and only if for \( \Omega = \{P\}, (c_1(\Omega),c_2(\Omega) \land d_1),(c_2(\Omega),c_2(\Omega) \land d_2) \ldots \) is accepted by \( A_{io}(P) \).

Proof. The termination of Algorithm 7.2 follows from the decidability of \( \equiv_c \) (Proposition 7.3.11), and the finiteness of the set of derivative modulo \( \equiv_c \) of \( P \) (Lemma 7.3.13) and of the set \( C(\Omega) \).

It follows from the construction that \( A_{io}(P) \) accepts \( (e_1,e_1'),(e_2,e_2')\ldots \) iff \( P = P_0 \xrightarrow{(e_1,e_1')} P_1 \xrightarrow{(e_2,e_2')} \ldots \) where each \( e_1,e_2\ldots \in C(\Omega) \). The result then follows Lemma 7.2.13. \( \square \)

The input-output automaton \( A_{io}(P) \) constructed by Algorithm 7.2 gives us a simple input-output execution model for \( P \in \text{Proc}^r \). We conclude this section with the description of such an execution model.
Execution of the automaton $A_{io}(P)$, $P \in \text{Proc}^\tau$. The execution starts at time unit 0 in the initial state of the automaton. At time $i$ the automaton is in some state $p$. Given an arbitrary input $c_i$ from the environment the automaton nondeterministically selects a transition from $p$ to some $q$ labeled by $(c_i(\Omega), c_i(\Omega) \land d_i)$, where $\Omega = \{P\}$. The constraint $d'_i = c_i \land d_i$ is the output to the environment at time $i$. The automaton then moves to state $q$ and repeats the same procedure (at time $i + 1$). We then say that on input $\alpha = c_1 c_2 \ldots$ the execution of the automaton can produce an output $\alpha' = d'_1 d'_2 \ldots$.

It follows from Lemma 7.3.17 that the input-output behavior of $P$ is exactly the set of all $(\alpha, \alpha')$ such that on input $\alpha$ the execution of $A_{io}(P)$ can produce $\alpha'$.

### 7.4 Summary and Related Work

In this chapter, we studied the relationship among the various ntcc equivalences (i.e., input-output, strongest-postcondition and output equivalence) and their induced congruences. In particular, we showed that although output equivalence is weaker than input-output equivalence, their induced congruences coincide for star-free processes.

We also gave a characterization of each ntcc equivalence (including output congruence) in terms of output equivalence over a particular family of contexts. Basically, the characterization of each ntcc equivalence, say $\sim$, reduces the question of whether $P \sim Q$ to whether $C_\sim[P] \sim_o C_\sim[Q]$ where $C_\sim$ is a particular context for which there is an effective construction, given $P$ and $Q$.

We showed that such a context is universal (i.e., the same for every $P$ and $Q$), if the underlying set of constraints $C$ is finite. If $C$ is not finite then the construction of the context involved a particularly interesting result: Given an arbitrary finite set of processes $\Omega$, one can construct a finite set of constraint $C(\Omega)$ containing all the relevant constraints for processes in such set. More precisely, $P \xrightarrow{(c, c \land d)} Q$ iff $P \xrightarrow{(c(\Omega), c(\Omega) \land d)} Q$ where $c(\Omega)$ is the strongest constraint in $C(\Omega)$ entailed by $c$.

Such a result extends that of [SJG96] which shows how to compute a strict subset of the relevant constraints of tcc processes by leaving out those arising from universal and existential quantification. This implies that such set can only characterize the behavior of hiding-free processes. In our case it was a key issue to compute the whole set in order to give all the characterizations above mentioned.

Furthermore, by using the standard theory of Büchi automata, we showed that the output equivalence is decidable for restricted-nondeterministic processes. Crudely speaking, these are star-free processes in which local processes must not exhibit nondeterministic behavior. This represents a substantial fragment on the ntcc processes. In fact, every process in the tcc model of [GJS96] and every application example in this thesis belong to this fragment.

From the decidability of the output equivalence and with the help of the
characterizations mentioned above it follows that all the \texttt{ntcc} equivalences are decidable for restricted-nondeterministic processes.

The work in [SJG96] shows how to compile a \textit{deterministic} extension of \texttt{tcc} into finite-state machines. Rather than a direct way of verifying process equivalences, such machines provide an execution model for \texttt{tcc} similar to the input-output automata execution model for \texttt{ntcc} shown in Section 7.3.5. In particular, the standard language equivalence between two automata defined as in our output automata construction implies the output equivalence of the processes they represent. This implication does not hold in general for the construction in [SJG96]. Notice also that for the \texttt{ntcc} case the main technical problems in trying to give a finite-state representation of processes arose exactly from allowing nondeterminism.

The decidability results for output equivalence, and the output and input-output congruences for \texttt{ntcc} were published as [NV02]. The result about the relevant set of constraints was published as [NPV02a]. The decidability results for the strongest postcondition and input-output equivalence have not been published.
Chapter 8

Variations of the Calculus: The TCC Hierarchy

In the literature there are several tcc languages differing in their way of expressing infinite behavior. The main purpose of this chapter is to study the expressive power of a few fundamental representatives of these languages by representing them as variants of the ntcc calculus. We do this by using automata theory, word problems, and adapting to our case a simulation technique from concurrency theory.

As such tcc languages are deterministic we shall confine our attention to the deterministic fragment of ntcc (see Definition 4.4.1).

We shall study in detail the following variants of the ntcc calculus:

- **rep**: deterministic ntcc; infinite behavior given by replication.

- **rec_p**: obtained from deterministic ntcc but instead of replication, it uses parametric recursion. In rec_p each procedure's body has no free variables other than its formal parameters.

- **rec_c**: same as rec_p, but where the actual parameters in recursive calls are identical to the formal parameters; i.e., we do not vary the parameters in the recursive calls.

- **rec_d**: obtained by using parameterless recursion, but with free variables in procedure bodies with dynamic scope.

- **rec_s**: same as rec_d but with static scope.

- **rec_0**: same as rec_p but recursion is given by parameterless recursion, hence without free variables in procedure bodies.

The expressive power of these process languages is compared relatively to the notion of input-output behavior studied in Chapter 7. Namely, one language is considered at least as expressive as another if the input-output behavior expressed by a process in the latter can be expressed also by a process in the former. Our comparison results can be summarized as follows:
• \textsc{rec}_p and \textsc{rec}_d are equally expressive, and strictly more expressive than the other tcc languages,

• \textsc{rep}, \textsc{rec}_a and \textsc{rec}_1 are equally expressive, and strictly more expressive than \textsc{rec}_0.

We shall actually show a strong separation result between \textsc{rec}_p/\textsc{rec}_d and \textsc{rep}/\textsc{rec}_a/\textsc{rec}_1. Namely, that input-output equivalence is undecidable for the languages in the first class but decidable for the languages in the second class for arbitrary constraint systems. The undecidability results holds even if we fix an underlying constraint system with a finite domain having at least one element. The undecidability result is obtained by a reduction from the Post’s correspondence problem [Pos46] and input-output preserving encoding between \textsc{rec}_p/\textsc{rec}_a. The decidability results follows from the Büchi automata [Buc62] representation of ntcc processes given in Chapter 7 and input-output preserving encoding between the languages in \textsc{rep}/\textsc{rec}_a/\textsc{rec}_1.

The expressiveness gaps illustrated above may look surprising to those acquainted with the \(\pi\)-calculus, because the \(\pi\)-calculus correspondents of \textsc{rep}, \textsc{rec}_1 and \textsc{rec}_p have all the same expressive power. The reason is that the \(\pi\)-calculus has some powerful mechanisms such as mobility which compensate for the weakness of replication and of the lower forms of recursion.

This chapter is structured as follows. The first section is devoted to describing the semantics of the various tcc languages. Section 8.2 states the relationship between the equivalences and their congruences for the various languages. Section 8.3 presents the undecidability of the input-output equivalence for \textsc{rec}_p processes in a finite-domain constraint system. In section 7.3 we state the decidability of the input-output equivalence for \textsc{rep} process over arbitrary constraint system. Section 8.5 states the classification (or hierarchy) of the tcc languages as stated above. In Section 8.6 we present encodings preserving the input-output semantics, and the classification of the tcc languages as stated above. Finally, in Section 8.7 we introduced a simulation technique for proving the input-output equivalence two processes, possibly belonging to different tcc language.

8.1 TCC Languages

In this section we describe the various languages. As mentioned above they differ in the ways of expressing infinite behavior through the time intervals.

8.1.1 Replication

We shall use \textsc{rep} to denote the deterministic fragment of ntcc introduced in Definition 4.4.1. Recall that processes in the deterministic fragment are those star-free processes in which the cardinality of every summation index set is at
most one. Thus, the resulting syntax of process in _rep_ is given by:

\[
P, Q, \ldots ::= \text{skip} \mid \text{tell}(c) \mid \text{when } c \text{ do } P \mid P \parallel Q \mid (\text{local} x) P \\
| \text{next } P \mid \text{unless } c \text{ next } P \mid \text{!} P
\]

(8.1)

Infinite behavior in _rep_ is provided by using replication. This way of expressing infinite behavior is also considered in [SJG96]. More precisely, [SJG96] uses the _hence_ operator. However, _hence_ \( P \) is equivalent to _next_ \( \text{!} P \) and, similarly \( \text{!} P \) is equivalent to \( P \parallel \text{hence} P \).

### 8.1.2 Recursion

An alternative to define infinite behavior in tcc languages is by using recursion as was done in [SJG94a, SJG94b, Tin99]. Consider the process syntax obtained from replacing in Equation 8.1 the replication \( \text{!} P \) with \( A(y_1, \ldots, y_n) \). More precisely,

\[
P, Q, \ldots ::= \text{skip} \mid \text{tell}(c) \mid \text{when } c \text{ do } P \mid P \parallel Q \mid (\text{local} x) P \\
| \text{next } P \mid \text{unless } c \text{ next } P \mid A(y_1, \ldots, y_n)
\]

(8.2)

The process \( A(y_1, \ldots, y_n) \) is an _identifier_ with arity \( n \). We assume that every identifier has a (recursive) _process (or procedure) definition_ of the form \( A(x_1, \ldots, x_n) \overset{\text{def}}{=} P \) where the \( x_i \)'s are pairwise distinct, and the intuition is that \( A(y_1, \ldots, y_n) \) behaves as \( P \) with \( y_i \) replacing \( x_i \) for each \( i \).

We declare \( \mathcal{D} \) to be the set of recursive definitions under consideration. We shall often use the notation \( \overline{x} \) as an abbreviation of \( x_1, x_2, \ldots, x_n \) if \( n \) is unimportant or obvious. We shall sometimes say that \( A(\overline{y}) \) is an _invocation_ with _actual parameters_ \( \overline{y} \), and given \( A(\overline{x}) \overset{\text{def}}{=} P \) we shall refer to \( P \) as its _body_ and to \( \overline{x} \) as its _formal parameters_.

### Finite Dependency and Guarded Recursion

Following [SJG94a], we shall require, for all the forms of recursion defined next, the following: (1) any process to depend only on finitely many definitions and (2) recursion to be “next” guarded. For example, given \( A(\overline{x}) \overset{\text{def}}{=} P \), every invocation \( A(\overline{y}) \) in \( P \) must occur within the scope of a “next” or “unless” operator. This avoids non-terminating sequences of internal reductions (i.e., non-terminating computation within a time interval). Below we give a precise formulation of (1) and (2).

Given \( A_1(\overline{x}_1) \overset{\text{def}}{=} P_1 \) and \( A_2(\overline{x}_2) \overset{\text{def}}{=} P_2 \), we say that \( A_1 \) (directly) _depends_ on \( A_2 \), written \( A_1 \leadsto A_2 \), if there is an invocation \( A_2(\overline{y}) \) in \( P_1 \). Requirement (1) can be then formalized by requiring the strict ordering induced by \( \leadsto^* \) (the reflexive and transitive closure of \( \leadsto \))\(^1\) to be well founded.

\(^1\)The relation \( \leadsto^* \) is a pre-ordering. By induced strict ordering we mean the strict component of \( \leadsto^* \) modulo the equivalence relation obtained by taking the symmetric closure of \( \leadsto^* \).
To formalize (2), suppose that $A_1 \leadsto A_2 \leadsto \ldots \leadsto A_n \leadsto A_{n+1} = A_1$, where $A_i(\bar{x}_i) := P_i$. We shall require that for at least one $i$, $1 \leq i \leq n$, the occurrences of $A_{i+1}$ in $P_i$ are within the scope of a “next” or an “unless” operator.

**Parametric Recursion**

We consider a further restriction for the case of recursion involving parameters. *All the free variables in definitions’ bodies must be formal parameters.* More precisely, for each $A(x_1, \ldots, x_n) := P$, decree that $f_{v}(P) \subseteq \{x_1, \ldots, x_n\}$. This requirement is imposed on the recursive versions of the $\pi$-calculus.

We shall use $\text{rec}_p$ to denote the tcc language with recursion with the above syntactic restriction. The operational rules for $\text{rec}_p$ are obtained from Table 3.1 by replacing to the rule for replication

\[
\text{REP} \quad \frac{}{\langle ! P, d \rangle \rightarrow \langle P \parallel \text{next} ! P, d \rangle}
\]

with the rule for recursion

\[
\text{REC} \quad \frac{}{\langle A(\bar{y}), d \rangle \rightarrow \langle P[\bar{y}/\bar{x}], d \rangle}
\]

As usual $P[y_1, \ldots, y_n/x_1, \ldots, x_n]$, with all the $x_i$’s being pairwise distinct, is the process that results from syntactically replacing every free occurrence of $x_i$ by $y_i$ using $\alpha$-conversion wherever needed to avoid capture.

**Identical Parameters Recursion.**

An interesting tcc language considered in [SJG94a] arises from $\text{rec}_p$ by restricting the parameters not to change through recursive invocations. In the $\pi$-calculus this restriction does not cause any loss of expressive power since such form of recursion can encode replication and replication can encode general recursion (see [Mil99]).

An example satisfying the above restriction is $R_P(\bar{x}) := P \parallel \text{next} R_P(\bar{x})$. Here the actual parameters of the invocation in the definition’s body are the same as the formal parameters of $R_P$. An example not satisfying the restriction is $R^\prime_P(\bar{x}) := P \parallel \text{next (local $\bar{x}$)} R^\prime_P(\bar{x})$. Here the actual parameters, although syntactically the same, are bound and therefore different from those of the formal parameters.

One can formalize the identical parameters restriction on a set of mutually recursive definitions as follows. Suppose that $A_1 \leadsto A_2$ and $A_2 \leadsto^* A_1$ with $A_1(\bar{x}_1) := P_1$ and $A_2(\bar{x}_2) := P_2$ in the underlying set of definitions $D$. Then for each invocation $A_2(\bar{y})$ in $P_1$ we should require $\bar{y} = \bar{x}_2$ and $\bar{y} \notin \text{bo}(P_1)$. In other words the actual parameters of the invocation $A_2$ in $P_1$ (i.e., $\bar{y}$) should be syntactically the same as its formal parameters (i.e., $\bar{x}_2$). Furthermore, they should not be bound in $P_1$ to avoid cases such as $R^\prime_P(\bar{x})$ above.

The processes of tcc with identical parameters are those of $\text{rec}_p$ that satisfy this requirement. We shall refer to this language as $\text{rec}_i$. 

8.1.3 Parameterless Recursion.

Tcc with parameterless recursion have been studied in [SJG94a]. We shall refer to identifiers with arity zero and their corresponding definitions as constant identifiers and constant definitions, respectively. We omit the “( )” in \( A( ) \).

Given a parameterless definition \( A \equiv P \), requiring all variables in \( f_0(P) \) to be formal parameters, as we did in \( \text{rec}_p \), would be too restrictive. This would mean that the body \( P \) has no free variables and processes in ccp communicate through free variables. For example, it would be impossible to define the process that every two time units tells \( x = 1 \). Consequently, let us consider a fragment allowing only parameterless recursion with free variables in the bodies of constant definitions.

Now assuming that the operational rules for parameterless recursion are the same as for \( \text{rec}_p \), one may wonder about the scope of the free variables in definitions bodies. Is it some kind of dynamic scoping similar to that of CCS [Mil89] and, most notably, as it is in the standard model of concurrent constraint programming [SRP91]? Is it static as in most programming languages?

The next section answers this question. Let us first illustrate what we mean by dynamic and static scoping.

**Example 8.1.1.** Consider a constant identifier \( A \) with the following definition

\[
A \equiv \begin{cases} \text{tell}(x = 1) & \\
\text{next} \text{(local} x \text{)} \ (A \ \text{when} \ x = 1 \ \text{do} \ \text{tell}(z = 1)) & 
\end{cases}
\]

In the case of dynamic scoping, an outside invocation \( A \) causes the execution \( \text{tell}(z = 1) \) in the second time interval. The reason is that \( \text{(local} x \text{)} \) binds the \( x \) resulting from the unfolding of the \( A \) inside the definition’s body\(^2\). In fact, the telling of \( x = 1 \), in the second time unit, will not be visible in the store. In the case of static scoping, \( \text{(local} x \text{)} \) does not bind the \( x \) of the unfolding of \( A \) because such an \( x \) is intuitively a “global” variable, and hence \( \text{tell}(z = 1) \) will not be executed. In fact, the telling of \( x = 1 \), will also be visible in the store in the second time interval. \( \square \)

**Parameterless Recursion with Dynamic Scoping**

The rule LOC in Table 3.1 combined with REC causes the parameterless recursion to have dynamic scoping\(^3\). As illustrated in the example below, the idea is that since \( \text{(local} x \text{)} P \) reduces to a process of the form \( \text{(local} x \text{)} Q \), the \( x \)’s occurring free in the unfolding of invocations in \( P \) get bounded.

**Example 8.1.2.** Let \( A \) as defined in Example 8.1.1. Let us abbreviate the definition of \( A \) as \( A \equiv \text{tell}(x = 1) \parallel P \). Also let \( Q = \text{skip} \parallel P \). We have the

\(^2\)Just as in the CCS definition \( A \equiv \alpha.0 \parallel \tau. (A \parallel \bar{\alpha}.0) \bar{\alpha} \), the process \( \bar{\alpha}.0 \) can communicate through \( \alpha \) with the unfolding of \( A \).

\(^3\)Rules LOC and REC are basically the same in ccp, hence the observations made in this section regarding dynamic scoping apply to ccp as well.
following reduction of $(\text{local} x) A$ in store $\text{true}$.

\[
\begin{align*}
(tell(x = 1), \text{true}) & \rightarrow \langle \text{skip}, x = 1 \rangle \\
(tell(x = 1) \parallel P, \text{true}) & \rightarrow \langle Q, x = 1 \rangle \\
(A, \text{true}) & \rightarrow \langle Q, x = 1 \rangle \\
(\langle \text{local} x, \text{true} \rangle \cdot A, \text{true}) & \rightarrow \langle (\text{local} x, x = 1) Q, \text{true} \rangle
\end{align*}
\]

Thus, $(\text{local} x) A$ in store $\text{true}$ reduces to $(\text{local} x, x = 1) (\text{skip} \parallel P)$ in store $\text{true}$. Notice that the free $x$ in $A$’s body become local to $(\text{local} x, x = 1) (\text{skip} \parallel P)$, i.e., it now occurs in the local store but not in the global one.

We shall refer to the language allowing only parameterless recursion with free-variables in the procedure bodies as $\text{rec}_d$; parameterless recursion with dynamic scoping.

Remark 8.1. It should be noticed that, unlike in $\text{rec}_p$, we cannot freely $\alpha$-convert processes in $\text{rec}_d$ without changing behavior. For example, we could $\alpha$-convert the $(\text{local} x) A$ in the above example into $(\text{local} z) A$ (since $A[z/x]$ is syntactically equal to $A$) but the behavior of $(\text{local} z) A$ would not be the same as that of $(\text{local} x) A$. We could solve this problem by defining the substitutions $[z/x]$ to be relabeling functions as in CCS instead of syntactic replacements. We can see in Table 3.1, however, that no syntactic substitutions will be applied in the reductions of $\text{rec}_d$ as this deals only with constant definitions. Therefore, the operational semantics in $\text{rec}_d$ does not appeal to $\alpha$-conversion.

Parameterless Recursion with Static Scoping

From the previous section it follows that if we want to have static scoping as in [SJG94a] we should replace the rule for local behavior $\text{LOC}$.

The rule $\text{LOC}$ defines locality for the parameterless recursion with static scoping language henceforth referred to as $\text{rec}_s$.

\[
\text{LOC'} \quad \langle P[y/x], d \rangle \rightarrow \langle P', d' \rangle \quad \text{if } y \text{ is fresh} \quad (8.4)
\]

As in [MPSS95], we use the notion of fresh variable meaning that it does not occur elsewhere in a process, definition or the store. It will be convenient to presuppose that the set of variables $V$ is partitioned into two infinite sets $F$ and $V - F$. We shall assume that the fresh variables are taken from $F$ and that no input from the environment or process, other than the ones generated when applying $\text{LOC}'$, can contain variables in $F$.

The fresh variables introduced by $\text{LOC}'$ are not to be visible from the outside. We hide these fresh variables, as it is done in [SJG96], by using existential quantification in the output constraint of observable transitions. More precisely, we replace in Table 3.1 the rule for the observable transitions $\text{OBS}$ with the rule

\[
\begin{align*}
\text{OBS'} \quad \langle P; c \rangle & \rightarrow^* \langle Q; d \rangle \quad \rightarrow
\\
P & \xrightarrow{(e, \exists x d)} F(Q) \quad (8.5)
\end{align*}
\]


where \( \exists_d \) represents the constraint resulting from the existential quantification in \( d \) of free occurrences of variables in \( \mathcal{F} \).

In order to see why LOC' causes static scoping in \( \text{rec}_a \), suppose that \( P \) in Rule LOC' in Equation 8.4 contains an invocation \( A \) where \( \bar{A} \overset{\text{def}}{=} R \). When replacing \( x \) with \( y \) in \( P \), \( A \) remains the same since \( A[y/x] \) is \( A \). Furthermore, since \( y \) is chosen from \( \mathcal{F} \), there will be no capture of free variables in \( R \) when unfolding \( A \). This causes the scoping to be static. Let us illustrate this by revisiting the previous example.

**Example 8.1.3.** Let \( A, P \) and \( Q \) as in the previous example. We have the following reduction of \( (\text{local} \ x) \ A \) in store \( \text{true} \).

\[
\begin{align*}
\langle \text{tell}(x = 1), \text{true} \rangle &\rightarrow \langle \text{skip}, x = 1 \rangle & \text{TELL} \\
\langle \text{tell}(x = 1) \parallel P, \text{true} \rangle &\rightarrow \langle Q, x = 1 \rangle & \text{PAR} \\
\langle A, \text{true} \rangle &\rightarrow \langle Q, x = 1 \rangle & \text{REC} \\
\langle (\text{local} \ x) A, \text{true} \rangle &\rightarrow \langle Q, x = 1 \rangle & \text{LOC'}
\end{align*}
\]

Thus, \( (\text{local} \ x) \ A \) in store \( \text{true} \) reduces to \( \text{skip} \parallel P \) in store \( (x = 1) \) making the free \( x \) in \( A \)'s body, as oppose to the previous example, visible in the “global” store.

**Remark 8.2.** Notice that, as in \( \text{rec}_d \), in \( \text{rec}_a \) we do not need \( \alpha \)-conversion since in the reductions of \( \text{rec}_a \) we only use syntactic replacements of variables by fresh variables.

### 8.1.4 Summary of TCC Languages

We described several languages based on the literature of (Timed) ccp. We have \( \text{rep} \) the tcc language with replication and \( \text{rec}_p \) the tcc language with recursion instead. A special case of \( \text{rec}_p \) is \( \text{rec}_i \) which restricts the parameters not to change through the recursive invocations. We also have the parameterless recursion languages \( \text{rec}_d \) and \( \text{rec}_a \). The former deals with dynamic-scoping while the later deals with static scoping.

For the sake of completeness, we consider here an additional language: \( \text{rec}_0 \), the language with neither parameters nor free variables in the bodies of definitions.

We adopt the following convention.

**Convention 8.1.4.** We shall use \( \mathcal{L} \) to designate the set of tcc languages

\[ \{ \text{rep}, \text{rec}_p, \text{rec}_i, \text{rec}_d, \text{rec}_a, \text{rec}_0 \} \]

Furthermore, we shall index sets and relations involving tcc processes with the appropriate tcc language name to make it clear what is the language under consideration. We shall omit the index when it is unimportant or clear from the context.

For example, \( \rightarrow_{\text{rec}_p} \) and \( \xrightarrow{\text{...}}_{\text{rec}_p} \) mean that the (internal and observable) reduction under considerations are those of \( \text{rec}_p \). Similarly, \( \text{Proc}_{\text{rec}_p} \) denotes the set of processes in \( \text{proc}_{\text{rec}_p} \), \( \sim_{\text{rec}_p} \) denotes the input-output equivalence (Definition 3.5.4) for processes in \( \text{Proc}_{\text{rec}_p} \), and \( \approx_{\text{rec}_p} \) denotes congruence induced by \( \sim_{\text{rec}_p} \) (Definition 7.1.4).
8.2 The TCC Equivalences

In this section we relate the equivalences and their congruences for the various tcc languages. Each behavioral equivalence (and congruence) for a tcc language $\ell$ is obtained by taking the tntcc transitions given in Definition 3.5.3 (and thus in Definitions 3.5.4 and 7.1.4) to be those of $\ell$ (i.e., replace $\xrightarrow{()}$ with $\xrightarrow{()}^\ell$).

The theorem below states the relationship among the equivalences.

**Theorem 8.2.1. (Equivalences' Relationship).** For each $\ell \in \mathcal{L}$,

1. If $\ell = \text{rec}_a$ then $\approx_{\text{io}}^\ell = \approx_0^\ell \sqsubseteq \sim_0^\ell \subseteq \sim_0^\ell$.

2. If $\ell \neq \text{rec}_a$ then $\approx_{\text{io}}^\ell = \approx_0^\ell = \sim_{\text{io}}^\ell \subseteq \sim_0^\ell$.

**Proof.** Let $\ell = \text{rec}_a$. Consider $\approx_{\text{io}}^\ell \subseteq \sim_{\text{io}}^\ell$. The inclusion is obvious. As for the proper inclusion, take $A \overset{\text{def}}{=} \text{tell}(c)$ with $c = (x = y)$ in any underlying constraint system with equality. One can verify that $A \sim_{\text{io}}^\ell \text{ tell}(c)$ but

$$(\text{local } x) A \sim_{\text{io}}^\ell \text{ tell}(c) \not\sim_{\text{io}}^\ell \text{ skip } \sim_{\text{io}}^\ell (\text{local } x) \text{ tell}(c).$$

The various cases for $\text{rep}$ follow from Theorem 7.1.5. The other cases follow from results in [SJG94a] and [SRP91].

The theorem says the input-output and output congruences coincide for all languages. It also states that the input-output behavior is a congruence for every tcc language but $\text{rec}_a$. As expected, it follows from the examples in the proof of Item (1) of the theorem above, that the input-output behavior of an arbitrary process $(\text{local } x) P$ in $\text{rec}_a$ cannot be inferred from the input-output behavior of $P$ only. This reveals a distinction between $\text{rec}_a$ and the other tcc languages and, in fact, between $\text{rec}_a$ and the standard model of concurrent constraint programming [SRP91].

In the following sections we shall classify the tcc languages based on the decidability of their input-output equivalence.

8.3 Undecidability Results

In this section we first state that $\sim_{\text{io}}^{\text{rec}_a}$ is undecidable for processes with an underlying finite-domain constraint system. Recall that a finite-domain constraint system $\text{FD}[n]$ (see Definition 3.1.4) provides a theory of variables ranging over a finite domain of values $D = \{0, 1, \ldots, n-1\}$ with syntactic equality over these values. We shall also prove a stronger version of this result establishing that $\sim_{\text{io}}^{\text{rec}_a}$ is undecidable even for the finite-domain constraint system with one single constant $\text{FD}[1]$, i.e., $|D| = 1$. In sections 8.6 we shall give an input-output preserving encoding from $\text{rec}_a$ into the parameterless recursion language $\text{rec}_a$. Therefore, $\sim_{\text{io}}^{\text{rec}_a}$ is undecidable as well.
8.3.1 Undecidability over Finite-Domains

Let us state our first undecidability result.

**Theorem 8.3.1. (Undecidability of $\sim^{\text{rec}}_{i0}$).** Given $P, Q \in \text{Proc}_{\text{rec}}$ in a finite-domain constraint system, the question of whether $P \sim^{\text{rec}}_{i0} Q$ or not is undecidable.

The proof of the theorem above will proceed by a reduction from the Post’s correspondence problem (PCP) [Pos46]. Let us recall the following definition.

**Definition 8.3.2 (PCP).** A Post’s Correspondence Problem (PCP) instance is a tuple $(V, W)$, where $V = \{v_0, \ldots, v_n\}$ and $W = \{w_0, \ldots, w_m\}$ are two sets of non-empty words over the alphabet $\{0, 1\}$. A solution to this instance is a sequence of indexes $i_0, \ldots, i_m$ in $I = \{0, \ldots, n\}$ with $i_0 = 0$ s.t.

$$v_{i_0}v_{i_1}\cdots v_{i_m} = w_{i_0}w_{i_1}\cdots w_{i_m}.$$

In the PCP we are given a PCP instance $(V, W)$ and we are asked whether there is a solution for such an instance.

The Post’s Correspondence Problem is known to be undecidable [Pos46]. In the next section we reduce PCP to the problem of deciding input-output equivalence between $\text{rec}_p$ processes, thus proving Theorem 8.3.1.

8.3.2 The Post’s Correspondence Problem Reduction

Let $(V, W)$ be a PCP instance where $V = \{v_0, \ldots, v_n\}$ and $W = \{w_0, \ldots, w_m\}$ are sets of non-empty words. Let $\text{FD}[m]$ (Definition 3.1.4) be the underlying constraint system where $m = \max(|V|, 2)$ (i.e., we need at least two constants in the encoding below).

For each $i \in I = \{0, \ldots, |V| - 1\}$, we shall define a process $A_i(a, b, index, x)$ which intuitively does the following:

1. It waits until is told that $a = 1$ to start writing $v_i$, one symbol per time unit. Each such a symbol, say $s$, will be written in $x$ by telling $x = s$. Similarly, it waits until $b = 1$ to start writing $w_i$, one symbol per time unit. Each such a symbol will also be written in $x$.

2. It spawns a process $A_j(a', b', index, x)$ when the environment inputs an index $index = j$ in $I$.

3. It sets $a = 0$ and $a' = 1$ when it finishes writing $v_i$, i.e., $|v_i|$ time units later after it started writing $v_i$ (this way it announces that its job of writing $v_i$ is done, and allows $A_j$ to start writing $v_j$). Similarly, it sets $b = 0$ and $b' = 1$ when it finishes writing $w_i$.

4. It aborts unless the environment provides an index in $I$. It also aborts if an inconsistency arises: Either two symbols (one from a $V$ word and another from a $W$ word) are written in $x$ in the same time unit and they do not match (thus generating false), or the environment itself inputs false.
Thus, intuitively the $A_i$’s keep writing $V$ and $W$ words, as the environment dictates, as long as the symbols match and the environment keeps providing indexes in $I$ at each time unit.

**Auxiliary Constructs** We use the following constructs:

$$W_{c, P}(\bar{x}) \overset{\text{def}}{=} \text{when } c \text{ do } P \text{ unless } c \text{ next } W_{c, P}(\bar{x})$$

$$R_Q(\bar{y}) \overset{\text{def}}{=} Q \text{ unless } R_Q(\bar{y})$$

where $fv(P) \cup fv(c) = \{\bar{x}\}$ and $fv(Q) = \{\bar{y}\}$. We use the more readable notation **wait $c$ do $P$** and **repeat $Q$** for $W_{c, P}(\bar{x})$ and $R_Q(\bar{y})$, respectively. We also define **whenever $c$ do $P$** as an abbreviation of **repeat when $c$ do $P$**.

We now define $A_i(a, b, index, x)$ for each $i \in I$ according to Items 1-4. The local variable $ichosen$ is used as flag to check whether the environment input an index.

$$A_i(a, b, index, x) \overset{\text{def}}{=} (\text{local } a' b' ichosen) (\text{wait } a = 1 \text{ do } V_i \text{ unless } \prod_{j \in I} \text{when } index = j \text{ do } (\text{tell}(ichosen = 1) \text{ unless } A_j(a', b', index, x)) \text{ unless } \text{Abort })$$

The process $V_i$ writes, one by one, the $v_i$ symbols in $x$ (notation $v_i(n)$ denotes the $n$–th element of $v_i$). Furthermore it sets $a = 0$ and $a' = 1$ when it finishes writing $v_i$. The process $W_i$ can explained analogously.

$$V_i = \prod_{0 \leq k < |v_i|} \text{next}^k \text{tell}(x = v_i(k)) \text{ unless } \text{next}^{|v_i|}(\text{tell}(a = 0) \text{ unless } \text{tell}(a' = 1))$$

$$W_i = \prod_{0 \leq k < |w_i|} \text{next}^k \text{tell}(x = w_i(k)) \text{ unless } \text{next}^{|w_i|}(\text{tell}(b = 0) \text{ unless } \text{tell}(b' = 1))$$

The process $\text{Abort}$ aborts, according to Item 4 above, by telling $\text{false}$ thereafter (thus creating a constant inconsistency).

$$\text{Abort} = \text{unless } ichosen = 1 \text{ next repeat } \text{tell}(\text{false}) \text{ unless } \text{false do repeat } \text{tell}(\text{false})$$

Let us now define a process $B_i(a, b, index, x, ok)$ for each $i \in I$ that behaves exactly like $A_i(a, b, index, x)$, but in addition it outputs $ok = 1$ whenever it stops writing $v_i$ and $w_i$ exactly in the same time interval. This happens when

---

*The reader may wonder why the $A_i$’s do not have the formal parameter $ok$ as well. This causes no problem here, but you can think of $A$ as having a dummy $ok$ formal parameter if you wish.*
both $a$ and $b$ are set to zero in the same unit and it will imply that a solution of the form $v_{i_0} \ldots v_i = w_{i_0} \ldots w_i$ for the PCP $(V,W)$ has been found.

$$B_i(a,b, \text{index}, x, ok) \overset{\text{def}}{=} (\text{local} a' b' \text{ ichosen}) (\text{wait } a = 1 \text{ do } V_i \parallel \text{wait } b = 1 \text{ do } W_i \parallel \prod_{j \in I} \text{when index } = j \text{ do } \text{tell} \text{ichosen } = 1 \parallel \text{next } B_j(a', b', \text{index}, x, ok))$$

$$\parallel \text{Abort} \parallel \text{whenever } a = 0 \land b = 0 \text{ do } \text{tell} \text{ok } = 1)$$

Since we require the first index in a solution for PCP $(V,W)$ to be 0, we define two processes $A(\text{index}, x)$ and $B(\text{index}, x, ok)$ which trigger $A_0$ and $B_0$ as follows.

$$A(\text{index}, x) \overset{\text{def}}{=} (\text{local } a \ b)(\text{tell}(a = 1) \parallel \text{tell}(b = 1) \parallel A_0(a, b, \text{index}, x))$$

$$B(\text{index}, x, ok) \overset{\text{def}}{=} (\text{local } a \ b)(\text{tell}(a = 1) \parallel \text{tell}(b = 1) \parallel B_0(a, b, \text{index}, x, ok))$$

One can verify that the only difference between a process $A(\text{index}, x)$ and $B(\text{index}, x, ok)$ is that the latter eventually tells $ok = 1$ iff there is a solution to the PCP $(V,W)$. More precisely,

**Lemma 8.3.3 (The PCP Reduction).** $A(\text{index}, x) \not\equiv_{io} B(\text{index}, x, ok)$ iff there is a solution to the PCP $(V,W)$.

**Proof.** The (rather lengthy) proof is given in Appendix A. 

Since the PCP problem is undecidable, from the lemma above it follows that given $P, Q \in \text{Proc}_{\text{rec}}$ in a finite-domain constraint system, the question of whether $P \sim_{io}^{\text{rec}} Q$ or not is undecidable. This proves Theorem 8.3.1.

### 8.3.3 Undecidability Over Fixed Finite-Domains

We now give a stronger version of the above theorem; input-output equivalence in undecidable in $\text{rec}$ even if we fix the underlying constraint system to be $\text{FD}[1]$, which is the finite-domain constraint system whose only constant is 0.

**Theorem 8.3.4. (Undecidability of $\sim_{io}^{\text{rec}}$ over $\text{FD}[1]$).** Fix $\text{FD}[1]$ to be the underlying constraint system. The question of whether $P \sim_{io}^{\text{rec}} Q$ or not is undecidable.

**Proof.** Let us consider the proof of Theorem 8.3.1. Let $m = \max(|V|, 2)$. Thus every value that a variable can take is in $D = \{0, \ldots, m - 1\}$. For each variable $y$ in the process definitions let us introduce $m$ new variables $y_0, \ldots, y_{m-1}$. Replace the declarations of $y$ (whether as a local or as formal parameter) by the corresponding declarations of the $y_0, \ldots, y_{m-1}$. Replace each constraint $y = v$ in the process definitions with $y_v = 0$. Create an inconsistency (i.e. by telling
false) whenever \( y_v = y_w \) for any two different values \( v \) and \( w \) in \( D \) (since \((y = v \land y = w) = \text{false})\).

We also have that the input-output and default output congruences are undecidable for \( \text{rec}_p \) over a fixed finite-domain constraint system.

**Theorem 8.3.5.** The input-output and output congruences \( \approx_{\text{io}}^{\text{rec}_p} \) and \( \approx_0^{\text{rec}_p} \) are undecidable for processes in the finite-domain constraint system \( \text{FD}[1] \).

**Proof.** Immediate from Theorems 8.3.4 and 8.2.1.

Notice that \( \text{FD}[1] \) is a very simple constraint system (i.e., only equality and one single constant). So, the undecidability results for other constraint systems providing theories with equality and an at least one constant symbol follow from Theorem 8.3.4. This includes almost all constraint system of interest (e.g., the Herbrand constraint system [Sar93], the Kahn constraint system [SRP91], Enumerated Types [Sar93] and modular arithmetic [PV01]).

### 8.4 Decidability Results

Unlike the \( \text{rec}_p \) case the input-output equivalence relation for \( \text{rep} \) is decidable even for arbitrary constraint systems. In fact, it follows from the results in Section 7.3 that all the equivalences we have considered in Definition 3.5.4 are decidable for \( \text{rep} \). This is stated in the following theorem.

**Theorem 8.4.1.** The following equivalences for processes in \( \text{rep} \) over arbitrary constraint system are decidable:

1. The input-output equivalence \( \approx_{\text{io}}^{\text{rep}} \), default output equivalence \( \approx_0^{\text{rep}} \) and strongest-postcondition equivalence \( \approx_\text{sp}^{\text{rep}} \).

2. The output congruences \( \approx_{\text{io}}^{\text{rep}} \) and \( \approx_0^{\text{rep}} \).

**Proof.** The decidability of the relations in Item 1 follows as an immediate consequence of Theorem 7.3.15 and Corollary 7.3.16 in Section 8.4. Such a theorem states that such equivalences are decidable for restricted-nondeterministic processes (Definition 7.3.1) over arbitrary constraint system. Every process in \( \text{rep} \) is restricted-nondeterministic.

The decidability of the relations in Item 2 follows as immediate consequence of Theorem 8.2.1.

In Section 8.6 we shall show via encodings that \( \text{rep} \), \( \text{rec}_1 \), \( \text{rec}_a \) have the same expressive power. We then conclude that the corresponding equivalences for \( \text{rec}_1 \) and \( \text{rec}_a \) are also decidable. These decidability results in \( \text{rep} \) with arbitrary constraint system are to be contrasted to the undecidability results in \( \text{rec}_p \) with the simple finite-domain constraint system \( \text{FD}[1] \).
8.5 Classification of the TCC Languages

In this section we discuss the relation between the various tcc languages, and we classify them on the basis of their expressive power.

Figure 8.1 shows the sub-language inclusions and the encodings preserving the input-output semantics between the various tcc versions. Classes I, II, III represent a partition based on the expressive power: two languages are in the same class if and only if they have the same expressive power. We will first discuss the separation results, and then the equivalences.

![Diagram showing the classification of TCC languages](image)

Figure 8.1: Classification of the various tcc languages: The tcc hierarchy.

Given the encodings, which will be provided in the next section, the separation between Classes II and III is already suggested by the results in Sections 8.4 and 8.3. From the proof of Theorem 8.3.1 it follows that \( \text{rec}_p \) is capable of expressing the "behavior" of Post’s correspondence problems, and hence clearly capable of expressing output behavior not accepted by Büchi automata. From Theorem 7.3.14, it follows that the output behavior of every process in \( \text{rep} \) can be represented as language accepted by a Büchi automata.

**Lemma 8.5.1.** There exists a process \( P \) in \( \text{rec}_p \) s.t., for every process \( Q \) in \( \text{rep} \), \( \text{io}(P) \neq \text{io}(Q) \).

**Proof.** We shall show that the input-output behavior of the process \( \text{Count}(a) \) in \( \text{rec}_p \) defined below is different from that of any \( \text{rep} \) process. Recall that the (default) output behavior of a given \( P, o(P) \), is the set of sequences that a process can output on input \( \text{true}^\omega \) (Definition 3.5.3).

For the purpose of this proof we take the liberty of allowing summations in \( \text{rec}_p \) (notice, however, that \( \text{Count}(a) \) is deterministic). Let \( c = (y = 1) \). Define the processes

\[
\begin{align*}
\text{Count}(a) & \overset{\text{def}}{=} (\text{local} a') (\text{wait} a = 1 \text{ do } \text{next}(\text{tell}(c) \parallel \text{tell}(a' = 1) \\
& \quad \parallel \text{un} \text{less } c \text{ next } \text{Count}(a')) \\
\text{Env}(a) & \overset{\text{def}}{=} (\text{tell}(a = 1) + \text{tell}(\text{true})) \parallel \text{un} \text{less } a = 1 \text{ next Env}(a) \\
\text{Sys}(a) & \overset{\text{def}}{=} \text{Env}(a) \parallel \text{Count}(a)
\end{align*}
\]
Intuitively, in $Sys(a)$ the process $Count(a)$ counts, by telling $c$'s, the number of $true$'s (plus 2) told by the environment $Env(a)$ before telling $(a = 1)$. More precisely, the output behavior of $Sys(a)$, $o(Sys(a))$, is

$$\{true^n.(a = 1).c^{n+2}.true^ω \mid n \geq 0\}.$$ 

One can verify that $α' \in o(Sys(a))$ iff $(α, α') \in io(Count(a))$ with $α = true^n.(a = 1).true^ω$ and $α' = true^n.(a = 1).c^{n+2}.true^ω$ for some $n$.

Let $P = Count(a)$. Let us suppose that $Q$ in $rep$ is such that $io(P) = io(Q)$. Hence, we must have $o(Env(a) \parallel Q) = \{true^n.(a = 1).c^{n+2}.true^ω \mid n \geq 0\}$. But according to Lemma 7.3.14, we would then be able to effectively construct a finite-state automaton recognizing the set $\{true^n.(a = 1).c^{n+2}.true^ω \mid n \geq 0\}$. This leads to a contradiction, since from simple arguments of automata theory it follows that such a construction is impossible.

The separation between Classes I and II, on the other hand, follows from the fact that without parameters or free variables the recursive calls cannot communicate with the external environment, hence in $rec_0$ a process can produce information on variables for a finite number of time intervals only. More precisely, we have the following result:

**Lemma 8.5.2.** There exists a process $P$ in $rep$ s.t., for every process $Q$ in $rec_0$, $io(P) \neq io(Q)$.

**Proof.** Let $P$ be the $rep$ process $tell(x = 1)$. Notice that $(true^ω,(x = 1)^ω) \in io(P)$. We want to show that for every $Q$ in $rec_0$, $(true^ω,(x = 1)^ω) \not\in io(Q)$.

In order to obtain a contradiction, let us assume that $Q$ in $rec_0$ is such that $(true^ω,(x = 1)^ω) \in io(Q)$.

Clearly, for any process $R$ if $R \xrightarrow{(true,x=1)} R'$ then $x$ must occur free in $R$, i.e., $x \in fv(R)$. From the assumption, $Q = Q_0 \xrightarrow{(true,x=1)} Q_1 \xrightarrow{(true,x=1)} \cdots$. We claim that there is $k \geq 0$ s.t., $x \not\in fv(Q_k)$, which leads us to a contradiction.

Given $R$, define the depth of $x$ in $R$, $n_x(R)$, as 1 plus the maximal depth of nesting of next operators in $R$ at which $x$ occurs free if $x \in fv(R)$, otherwise decree that $n_x(R) = 0$. For example,

- $n_x(tell(y = 1)) = 0$
- $n_x(tell(x = 1)) = 1$
- $n_x(tell(x = 1) \parallel next \: tell(x = 1) \parallel next^3(\text{local } x) \: tell(x = 1)) = 2$

By using structural induction, we can easily verify that for all $R$ in $rec_0$ if $R \xrightarrow{(c,d)} R'$ and $n_x(R) > 0$ then $n_x(R) > n_x(R')$, thus proving the claim above.

### 8.6 The Encodings

This section is devoted to showing the encodings of the various tcc languages. Henceforth, $[\cdot] : \ell \rightarrow \ell'$ will represent the encoding function for each pair $\ell$
and \( \ell' \). We shall say that \([\cdot]\) is homomorphic wrt to the parallel operator if 
\([P \parallel Q] = [P] \parallel [Q]\), and similarly for the other operators.

**Notation 8.6.1.** We shall use the following notation:

- We use \( \text{call}(x) \) as abbreviation of \( x = 1 \) and declare, for each identifier \( A \), a fresh variable \( z_A \) uniquely associated to it.
- We denote by \( I(P) \) the set of identifiers \( P \) depends upon, i.e. the transitive closure of \( \sim \) of the identifiers occurring in \( P \) (see Section 8.1.2).
- We often use \( D_\ell \) to denote the set of recursive definitions under consideration for processes in \( \ell \). As usual we omit \( \ell \) when it is clear from the context.

### 8.6.1 Encoding \( \text{rec}_a \rightarrow \text{rep} \)

Here the idea is to simulate a procedure definition by a replicated process that activates (the encoding of) its body \( P \) each time there is a call for it. The activation can be done by using a construct of the form \textbf{when} c \textbf{do} P. The call, of course, will be simulated by \( \text{tell}(c) \).

The key case is the local operator, since we do not want to capture the free variables in the bodies of procedures. Thus, we need to \( \alpha \)-convert by renaming the local variables with fresh variables.

First we need two auxiliary encodings \([\cdot]_D \) and \([\cdot]_0 \) given by:

\[
[A \overset{\text{def}}{=} P]_D = \text{!when call}(z_A) \text{ do } [P]_0
\]

\[
[A]_0 = \text{tell}(\text{call}(z_A))
\]

\[
[(\text{local} x) P]_0 = (\text{local} y) ([P[y/x]]_0)
\]

where \( y \) is fresh

with \([\cdot]_0 \) being homomorphic on all the other operators of \( \text{rec}_a \).

We are now ready to give our encoding of \( \text{rec}_a \) into \( \text{rep} \).

**Definition 8.6.2.** The encoding \([\cdot] : \text{rec}_a \rightarrow \text{rep} \) is given by:

\[
[A] = (\text{local} \ z) ([P]_0 \parallel \prod_{i=1}^n [A_i(\bar{x}_i) \overset{\text{def}}{=} P_i]_D)
\]

with \( I(P) = \{A_1, \ldots, A_n\} \) and \( z = z_A \ldots z_A \).

### 8.6.2 Encoding \( \text{rec}_i \rightarrow \text{rep} \)

This encoding is similar to the encoding in the previous section, except that now we need to encode the passing of parameters as well. Let us give some intuition first.

A call \( A(y) \), where \( A(x) \overset{\text{def}}{=} P \), can occur in a process or in the definition of identifier \( B \) (possibly \( A \) itself). Consider the case in which there is no mutual
dependency between $A$ and $B$ or $A$ is a call in a process. Then, the actual parameters of $A$ may be different from the formal ones (i.e., $\bar{y} \neq \bar{x}$). If so, we need to model the call by providing a copy of the replicated process that encodes the definition of $A$ and by making the appropriate parameter replacements.

Now, consider the case in which there is a mutual dependency between $A$ and $B$ (i.e., if also $A$ depends on $B$). From the restriction imposed on (the mutual) recursion of $\text{rec}_1$ (see Section 8.1.2), we know that the actual parameters must coincide with the formal ones (i.e., $\bar{y} = \bar{x}$) and therefore we do not need to make any parameter replacement. Neither do we need to provide a copy of the replicated processes as it will be available at the top level.

As for the previous encoding, we first define the auxiliary encodings $\llbracket \cdot \rrbracket_D$ and $\llbracket \cdot \rrbracket_0$:

$$\llbracket A(\bar{x}) \rrbracket_D \overset{\text{def}}{=} \begin{cases} \text{!when call}(z_A) \text{ do } \llbracket P \rrbracket_0 & \text{if } \bar{y} = \bar{x} \text{ and } A(\bar{x}) \overset{\text{def}}{=} P \in D \\ \llbracket P \rrbracket_0 & \text{if } \bar{y} \neq \bar{x} \text{ and } A(\bar{x}) \overset{\text{def}}{=} P \in D \end{cases}$$

with $\llbracket \cdot \rrbracket_0$ homomorphism on all the other operators of $\text{rec}_1$.

It worth noticing that if we did not have the restriction on the recursion in $\text{rec}_1$ mentioned above, the encoding $\llbracket \cdot \rrbracket_D$ would not be well-defined. E.g., consider the definition $A(x) \overset{\text{def}}{=} \text{next} (\text{local } y) A(y)$ which violates the restriction, and try to compute $\llbracket A(x) \rrbracket_D = \llbracket (\text{local } y) A(y) \rrbracket_D$.

We are now ready to give our encoding of $\text{rec}_1$ into $\text{rep}$.

**Definition 8.6.3.** The encoding $\llbracket \cdot \rrbracket : \text{rec}_1 \rightarrow \text{rep}$ is given by:

$$\llbracket A(\bar{y}) \rrbracket = (\text{local } \bar{z}) \left( [P]_0 \parallel \prod_{i=1}^{n} [A_i(\bar{x}_i) \overset{\text{def}}{=} P_i]_D \right)$$

with $I(P) = \{ A_1, \ldots, A_n \}$ and $\bar{z} = z_{A_1} \ldots z_{A_n}$.

### 8.6.3 Encoding $\text{rep} \rightarrow \text{rec}_1$

This encoding is rather simple. The idea is to replace $!P$ by a call to a new process identifier $R_P$, defined as a process that expands $P$ and then calls itself recursively in the next time interval. The free variables of $!P$, $\bar{x}$, are passed as (identical) parameters.

**Definition 8.6.4.** The encoding $\llbracket \cdot \rrbracket : \text{rep} \rightarrow \text{rec}_1$ is given by:

$$\llbracket P \rrbracket = R_P(\bar{x})$$

where $R_P(\bar{x}) \overset{\text{def}}{=} [P] \parallel \text{next } R_P \in D_{\text{rec}_1}, \bar{x} = \text{fv}(P)$.

with $\llbracket \cdot \rrbracket$ homomorphism on all the other operators of $\text{rep}$.  


8.6.4 Encoding \( \text{rec}_d \to \text{rec}_p \)

Intuitively, if the free variables are treated dynamically, then they could equivalently be passed as actual parameters, identical to the formal ones.

**Definition 8.6.5.** The encoding \( [\cdot] : \text{rec}_d \to \text{rec}_p \) is given by

\[
[A] = A(\bar{x})
\]

where \( A \overset{\text{def}}{=} P \in \mathcal{D}_{\text{rec}_d} \)

and \( A(\bar{x}) \overset{\text{def}}{=} [P] \in \mathcal{D}_{\text{rec}_p}, \bar{x} = \text{fv}(P) \)

with \( [\cdot] \) homomorphic on all the other operators of \( \text{rec}_d \).

8.6.5 Encoding \( \text{rec}_p \to \text{rec}_d \)

The idea is to establish the link between the formal parameters \( \bar{x} \) and the actual parameters \( \bar{y} \) by telling the constraint \( \bar{x} = \bar{y} \). However, this operation has to be encapsulated within a (local \( \bar{x} \)) in order to avoid confusion with other potential occurrences of \( \bar{x} \) in the same context of the call.

**Definition 8.6.6.** The encoding \( [\cdot] : \text{rec}_p \to \text{rec}_d \) is given by

\[
[A(\bar{y})] = (\text{local} \bar{x})(A \parallel E_{\bar{y}/\bar{x}})
\]

where \( A(\bar{x}) \overset{\text{def}}{=} P \in \mathcal{D}_{\text{rec}_p}, \ A \overset{\text{def}}{=} [P] \in \mathcal{D}_{\text{rec}_d}, \)

and \( E_{\bar{y}/\bar{x}} \overset{\text{def}}{=} \text{tell}(\bar{y} = \bar{x}) \parallel \text{next} E_{\bar{y}/\bar{x}} \in \mathcal{D}_{\text{rec}_d} \)

with \( [\cdot] \) homomorphic on all the other operators of \( \text{rec}_d \).

8.6.6 Encoding \( \text{rep} \to \text{rec}_s \)

Here we take advantage of the automata representation of the input-output behavior of \( \text{rep} \) processes given by Algorithm 7.2 in Section 7.3.5. Basically, the idea is to use the recursive definitions as equations describing these input-output automata.

Let \( P \) be an arbitrary process in \( \text{rep} \). Let \( M_P = A^P_P \) be the automaton given by Algorithm 7.2 representing the input-output behavior of \( P \) on the inputs of relevance for \( P, C(\{P\}) \) (Definition 7.2.11).

Recall that the start state of \( M_P \) is \( P \). Let \( T_P \) be the set of transitions of \( M_P \). Each transition from \( Q \) to \( R \) with label \( c,d \), written \( \langle Q, (c,d), R \rangle \in T_P \), represents an observable transition \( Q \xrightarrow{(c,d)} R \), where \( c \in C(\{P\}) \).

For each state \( Q \) of \( M_P \) we define an identifier \( A_Q \) as follows:

\[
A_Q \overset{\text{def}}{=} \prod_{\langle Q, (c,d), R \rangle \in T_P} \text{when } c \ (\text{tell}(d) \parallel O(\sqcup c, R))
\]

with \( \sqcup c = \bigvee_{c \in \{c' \mid c' \notin c, c' \sqsubseteq c, \langle Q, (c',d'), R' \rangle \in T_P} e \)

where \( O(\sqcup c, R) \) takes the form \( \text{unless } \sqcup c \text{ next } A_R \) if \( c \neq \text{false} \), otherwise it takes the form \( \text{next } A_R \).

Intuitively, \( A_Q \) expresses that if we are in state \( Q \) and \( c \) is the strongest constraint entailed by the input, then the next state will be \( R \) and the output will be \( d \), with \( \langle Q, (c,d), R \rangle \in T_P \).
Definition 8.6.7. The encoding \([\cdot] : \text{rep} \to \text{rec}_s\) is defined as \([P] = A_P\).

8.7 Correctness of the Encodings Using a Simulation Technique

The following theorem states that the encodings defined in previous sections are all correct wrt the input-output behavior.

Theorem 8.7.1. For every encoding \([\cdot] : \ell \to \ell'\) defined from Section 8.6.1 through Section 8.6.6, we have \(io(P) = io([P])\).

Proof. The proof of correctness of each encodings is given in Appendix B. \(\square\)

Although, the various proofs of correctness of the encodings are given in the appendix, we wish to show in this section the main proof technique used in the proofs. The relevance of this technique is that we adapt to our case (i.e., tcc languages and ntcc), the notion of (bi)simulation [Mil89] which is central to concurrency theory.

8.7.1 A Simulation Proof Technique

The notion of simulation introduced in this section will allow us to compare the input-output behavior between two processes (possibly belonging to different tcc languages). Crudely speaking, the idea is that to prove that the input-output behavior of \(P\) can also be exhibited by \(Q\), we can provide a simulation relation including these processes. Such a simulation represents a sort of invariant holding between \(P\) and \(Q\).

Notation 8.7.2. We shall use the following notations and conventions.

- We take the liberty of writing \(P \sim_\omega Q\), where \(P\) is in \(\ell\) and \(Q\) is in \(\ell'\), to mean \(io_\ell(P) = io_{\ell'}(Q)\). It should be clear from the context the tcc languages \(\ell\) and \(\ell'\) under consideration. This will apply analogously to other process equivalences.

- Notation \(\Gamma_\ell\) denotes the set of configurations (i.e., process-store pairs) in the tcc language \(\ell\).

- Suppose that \(S\) and \(S'\) are binary relations. We often write \(aSb\) to denote \((a,b) \in S\). The composition \(SS'\) denotes the set of all \((a,b)\) s.t., \(aScS'b\) for some \(c\). We use \(S^{-1}\) to denote the set \(\{(b,a) \mid (a,b) \in S\}\), i.e., the converse of \(S\).

At this point, it is convenient to recall some previous notation: \(\exists x e\) denotes the constraint that results from the existential quantification of all fresh variables in \(e\) (Section 8.1.3). The function \(F(.)\) denotes the future function (Definition 3.3.10).

We need to consider upward closed relations on configuration wrt to information in the store.
Definition 8.7.3 (Store Upward Closure). Let $S \subseteq \Gamma_\ell \times \Gamma_{\ell'}$. The relation $S$ is store upward-closed iff $\langle P, c \rangle S \langle Q, d \rangle$ implies $\langle P, c \land e \rangle S \langle Q, d \land e \rangle$ for all $e \in C$ s.t. $e$ has no free occurrences of fresh variables.

Let us now introduce a notion of strong simulation and strong similarity. These notions will allow us to compare a process (configuration) in a tec language $\ell$ with a process (configuration) in a tec language $\ell'$.

Definition 8.7.4 (Strong Simulation & Strong Similarity). A store upward-closed relation $S \subseteq \Gamma_\ell \times \Gamma_{\ell'}$ is a strong simulation iff whenever $\langle P, c \rangle S \langle Q, d \rangle$,  

1. $\exists \gamma d \models \exists xc$,  
2. If $\langle P, c \rangle \rightarrow_\ell \gamma$ then there exists $\gamma' s.t., \langle Q, d \rangle \rightarrow_{\ell'} \gamma'$ and $\gamma S \gamma'$,  
3. If $\langle P, c \rangle \not\rightarrow_\ell$ then $\langle Q, d \rangle \not\rightarrow$ and $\langle F(P), \text{true} \rangle S \langle F(Q), \text{true} \rangle$.

We say that $\gamma'$ strongly simulates $\gamma$ iff $(\gamma, \gamma') \in S$ for some strong simulation $S$. Furthermore, if $\gamma'$ and $\gamma$ strongly simulate each other, we say that they are strongly similar, written $\gamma \sim \gamma'$. We shall refer to $\sim$ as strong similarity.

Intuitively, $\langle Q, d \rangle$ can simulate $\langle P, c \rangle$ iff  
1. their observable information match; $\exists xc = \exists xd$ (i.e., information about fresh variables is not observable)  
2. if there is an internal evolution from $\langle P, c \rangle$ then there is some internal evolution of $\langle Q, c \rangle$ that simulates it,  
3. if there is no internal evolution of $\langle P, c \rangle$ then there must not be internal evolution of $\langle Q, c \rangle$ and the external evolution of $\langle P, c \rangle$ (i.e., $\langle F(P), \text{true} \rangle$) must be simulated by the external evolution of $\langle Q, d \rangle$ (i.e., $\langle F(Q), \text{true} \rangle$).

We need simulations to be store upward-closed to capture quantification over all possible augmentations of the store by the environment. The environment cannot add information about the fresh variables. This is why we require $e$ in Definition 8.7.3 not to contain free occurrences of fresh variables.

Example 8.7.5. Let $\ell$ be an arbitrary tec language. We leave it to reader to verify that $\gamma = \langle P \parallel Q, c \rangle \sim \langle Q \parallel P, c \rangle = \gamma'$, where $\gamma, \gamma' \in \Gamma_\ell$, by establishing that the store upward-closures of  

$$B = \{ ((P_1 \parallel P_2, c), (P_2 \parallel P_1, c)) \mid P_1, P_2 \text{ in } \ell \}$$

and $B^{-1}$ are simulations.

At this point we could also introduce in the standard way the notion of bisimulation [Mil89] by requiring both $S$ and $S^{-1}$ to be simulations. Nevertheless, the notion of simulation is sufficient for our technical purposes.

The following proposition states that strong similarity is an equivalence relation on the set of all configurations. More precisely,
Proposition 8.7.6. Let $\gamma_1 \in \Gamma_{t_1}$, $\gamma_2 \in \Gamma_{t_2}$ and $\gamma_3 \in \Gamma_{t_3}$ be arbitrary configurations:

1. (Reflexivity) $\gamma_1 \sim \gamma_1$.

2. (Transitivity) if $\gamma_1 \sim \gamma_2$ and $\gamma_2 \sim \gamma_3$ then $\gamma_1 \sim \gamma_3$.

3. (Symmetry) if $\gamma_1 \sim \gamma_2$ then $\gamma_2 \sim \gamma_1$.

Proof. (1) The identity on any $\Gamma_t$ is a simulation. (2) If $S \subseteq \Gamma_t \times \Gamma_t$ and $S' \subseteq \Gamma_{t'} \times \Gamma_{t'}$ are simulations then $SS'$ is a simulation. (3) Immediate from the definition.

Let us now give the “weak” version of the previous definition by allowing the simulating configuration to move zero or more steps.

Definition 8.7.7 (Weak Simulation and Similarity). A store upward-closed relation $S \subseteq \Gamma_t \times \Gamma_t$ is a weak simulation iff whenever $\langle P, c \rangle S \langle Q, d \rangle$,

1. $\exists \exists x, \exists y$ s.t.

2. If $\langle P, c \rangle \rightarrow_{\Gamma} \gamma$ then there exists $\gamma'$ s.t., $\langle Q, d \rangle \rightarrow_{\Gamma}^{\ast} \gamma'$ and $S \gamma'$. 

3. If $\langle P, c \rangle \rightarrow_{\Gamma} \gamma$ then there exists $Q'$ s.t., $\langle Q, d \rangle \rightarrow_{\Gamma}^{\ast} \langle Q', d \rangle \rightarrow$ and $\langle F(P), \text{true} \rangle S \langle F(Q'), \text{true} \rangle$.

We say that $\gamma'$ (weakly) simulates $\gamma$ iff $(\gamma, \gamma') \in S$ for some weak simulation $S$. Furthermore, if $\gamma'$ and $\gamma$ weakly simulate each other, we say that they are (weakly) similar, written $\gamma \sim_w \gamma'$. We shall refer to $\sim_w$ as weak similarity.

As strong similarity, weak similarity is an equivalence in the set of all configurations. More precisely,

Proposition 8.7.8. Let $\gamma_1 \in \Gamma_{t_1}$, $\gamma_2 \in \Gamma_{t_2}$ and $\gamma_3 \in \Gamma_{t_3}$ be arbitrary configurations:

1. (Reflexivity) $\gamma_1 \sim_w \gamma_1$.

2. (Transitivity) if $\gamma_1 \sim_w \gamma_2$ and $\gamma_2 \sim_w \gamma_3$ then $\gamma_1 \sim_w \gamma_3$.

3. (Symmetry) if $\gamma_1 \sim_w \gamma_2$ then $\gamma_2 \sim_w \gamma_1$.

Proof. Similar to the proof of Proposition 8.7.8.

Obviously, strong similarity implies weak similarity but not the reverse.

Example 8.7.9. One can verify that $\langle \text{when true do } P, c \rangle \sim_w \langle P, c \rangle$, but $\langle \text{when true do } P, c \rangle \not\sim \langle P, c \rangle$.

Proposition 8.7.10. If $\gamma_1 \sim \gamma_2$ then $\gamma_1 \sim_w \gamma_2$.

Proof. Every strong simulation is a weak one.
The following two propositions make it easier to establish that two processes are weakly similar by allowing configurations in a simulation to be quotiented by weak bisimilarity. We first take the simulating configurations up to weak bisimilarity.

**Proposition 8.7.11.** The relation $S \sim_w$ is a weak simulation if the following condition holds: whenever $\langle P, c \rangle S \langle Q, d \rangle$,

1. $\exists x d \models \exists x c$,

2. If $\langle P, c \rangle \rightarrow_\ell \gamma$ then there exists $\gamma'$ s.t., $\langle Q, d \rangle \rightarrow^* \gamma'$ and $\gamma S \sim_w \gamma'$,

3. If $\langle P, c \rangle \not\rightarrow_\ell$ then there exists $Q'$ s.t., $\langle Q, d \rangle \rightarrow^*_\ell \langle Q', d \rangle \not\rightarrow$ and $(F(P), \text{true}) S \sim_w (F(Q'), \text{true})$.

**Proof.** We shall show that $S \sim_w$ satisfies Items 1-3 in Definition 8.7.7 are satisfied. Suppose that $\langle P, c \rangle S \sim_w \langle Q, d \rangle$ and that the condition in the proposition holds. Let $\langle Q_1, d_1 \rangle$ be such that $\langle P, c \rangle S \langle Q_1, d_1 \rangle \sim_w \langle Q, d \rangle$.

We first prove Item 1 in Definition 8.7.7. From the condition in the proposition it follows that $\exists x d_1 \models \exists x c$. Since $\langle Q, d \rangle$ simulates $\langle Q_1, d_1 \rangle$ we must have $\exists x d \models \exists x d_1$. Therefore, $\exists x d \models \exists x c$ as required.

To prove Item 2 in Definition 8.7.7, suppose that $\langle P, c \rangle \rightarrow_\ell \gamma$. From the condition in the proposition there exists $\gamma_1$ s.t., $\langle Q_1, d_1 \rangle \rightarrow^*_\ell \gamma_1$ and $\gamma S \sim_w \gamma_1$. But since $\langle Q, d \rangle$ and $\langle Q_1, d_1 \rangle$ simulate each other, then there must be a $\gamma'$ s.t., $\langle Q, d \rangle \rightarrow^*_\ell \gamma'$ and $\gamma_1 \sim_w \gamma'$. Therefore $\gamma S \sim_w \gamma_1 \sim_w \gamma'$ as required.

To prove Item 3 in Definition 8.7.7, suppose that $\langle P, c \rangle \not\rightarrow_\ell$. From Item 3 in the proposition there must exist $Q'_1$ s.t., $\langle Q_1, d_1 \rangle \rightarrow^*_\ell \langle Q'_1, d_1 \rangle \not\rightarrow$ and $(F(P), \text{true}) S \sim_w (F(Q'_1), \text{true})$. But since $\langle Q, d \rangle$ and $\langle Q_1, d_1 \rangle$ simulate each other, then there must be a $Q'$ s.t., $\langle Q, d \rangle \rightarrow^*_\ell \langle Q', d \rangle \not\rightarrow$ and $(F(Q'_1), \text{true}) \sim_w (F(Q'), \text{true})$. Hence, $(F(P), \text{true}) \sim_w (F(Q'_1), \text{true}) \sim_w (F(Q'), \text{true})$ as wanted.

We now consider the simulated configurations up to weak bisimilarity.

**Proposition 8.7.12.** The relation $\sim_w S$ is a weak simulation if the following condition holds: whenever $\langle P, c \rangle S \langle Q, d \rangle$,

1. $\exists x d \models \exists x c$,

2. If $\langle P, c \rangle \rightarrow^*_\ell \gamma$ then there exists $\gamma'$ s.t., $\langle Q, d \rangle \rightarrow^*_\ell \gamma'$ and $\gamma \sim_w S \gamma'$,

3. If $\langle P, c \rangle \not\rightarrow^*_\ell$ then there exists $Q'$ s.t., $\langle Q, d \rangle \rightarrow^*_\ell \langle Q', d \rangle \not\rightarrow$ and $(F(P), \text{true}) \sim_w S (F(Q'), \text{true})$.

**Proof.** Similar to the proof of Proposition 8.7.11.

**Remark 8.3.** Notice that unlike Proposition 8.7.11, in the second item of Proposition 8.7.12, we use $\rightarrow^*_\ell$ rather than $\rightarrow^*_\ell$. The reader can verify that if we had used $\rightarrow_\ell$ instead we could wrongly conclude that the configurations $\langle \text{when true do tell(c), true} \rangle$ and $\langle \text{skip, true} \rangle$ are weakly bisimilar.
Chapter 8. Variations of the Calculus: The TCC Hierarchy

The following theorem states that the notion of (weak) similarity can certainly help us proving input-output equivalence.

**Theorem 8.7.13 (Similarity & IO Equivalence).** \(\langle P, \text{true} \rangle \sim_w \langle Q, \text{true} \rangle\) implies \(P \sim_{io} Q\).

**Proof.** Suppose that \(\langle P, \text{true} \rangle \sim_w \langle Q, \text{true} \rangle\). Here we only illustrate the case \(io(P) \subseteq io(Q)\) as the case \(io(Q) \subseteq io(P)\) is symmetric.

Let \((\alpha, \alpha') \in io(P)\) with \(\alpha = c_1.c_2\ldots\) and \(\alpha' = c'_1.c'_2\ldots\) We want to prove that \((\alpha, \alpha') \in io(Q)\).

From the operational semantics, we must have

\[
P = P_1 \xrightarrow{(c_1, e_1)} P_2 \xrightarrow{(c_2, e_2)} \ldots
\]

where for \(i \geq 1\), \(\langle P_i, c_i \rangle \rightarrow^* \langle P'_i, d_i \rangle \not\rightarrow\) and \(P_{i+1} = F(P'_i)\) and \(c'_i = \exists x d_i -\) recall that fresh variables do not occur free in the inputs or the outputs.

From our assumption, there are two simulations \(S\) and \(S'\) containing the pairs \((\langle P, \text{true} \rangle, \langle Q, \text{true} \rangle)\) and \((\langle Q, \text{true} \rangle, \langle P, \text{true} \rangle)\), respectively. Because \(S\) and \(S'\) are store upward-closed, we must have \((\langle P, c_1 \rangle, \langle Q, c_1 \rangle) \in S\) and \((\langle Q, c_1 \rangle, \langle P, c_1 \rangle) \in S'\).

Since \(S\) is a simulation and \(\langle P_1, c_1 \rangle \rightarrow^* \langle P'_1, d_1 \rangle \not\rightarrow\), for \(Q = Q_1\) we should have a configuration \(\langle Q'_1, e_1 \rangle\) such that \(\langle Q_1, c_1 \rangle \rightarrow^* \langle Q'_1, e_1 \rangle \not\rightarrow\) with \(\exists x e_1 \models \exists x d_1\) and \((\langle F(P'_1), \text{true} \rangle, \langle F(Q'_1), \text{true} \rangle) \in S\). Since \(S'\) is a simulation and \(\langle Q_1, c_1 \rangle \rightarrow^* \langle Q'_1, e_1 \rangle \not\rightarrow\), we must have \(\langle P_1, c_1 \rangle \rightarrow^* \langle P'_1, d'_1 \rangle \not\rightarrow\), with \(\exists x d'_1 \models \exists x e_1\). But \(\exists x d'_1 = \exists x d_1\) since the tcc languages under consideration are deterministic (i.e., \(P\) on input \(c_1\) must give the always the same output).

Therefore \(\exists x e_1 = \exists x d_1\). Since \(c'_1 = \exists x d_1\), we have \(Q = Q_1 \xrightarrow{(c_1, e_1)} Q_2\) with \(Q_2 = F(Q'_1)\).

By applying the argument above repeatedly, we obtain a sequence

\[
Q = Q_1 \xrightarrow{(c_1, d_1)} Q_2 \xrightarrow{(c_2, d_2)} \ldots
\]

Therefore, \((\alpha, \alpha') \in io(Q)\), as desired. \(\square\)

It follows from the theorem above that in order to prove input-output equivalence between \(P\) and \(Q\) (possibly in different tcc languages), it suffices to exhibit two simulations: One including \((\langle P, \text{true} \rangle, \langle Q, \text{true} \rangle)\) and the other including \((\langle Q, \text{true} \rangle, \langle P, \text{true} \rangle)\). In fact, this is the main technique used in the various proofs of correctness wrt to the input-output behavior of the tcc encodings (see Appendix B).

### 8.8 Summary and Related Work

In this chapter we studied the expressive power of several tcc languages focusing on the decidability of their behavioral equivalences. We achieved this by using and adapting standard tools from automata theory (Büchi automata), words problems (The PCP problem) and concurrency theory (simulation techniques).
8.8. Summary and Related Work

In particular, we have shown that rep (i.e. tcc with replication) can be compiled into finite-state automata, while recp (i.e.tcc with recursion) cannot, not even when the constraint system is based on a fixed finite domain. Furthermore, we have presented behavior-preserving encodings between rep, rec1 (tcc with identical parameters recursion) and recs (tcc with parameterless recursion and static-scope free variables), and between rep and rec4 (tcc with dynamic-scope free variables). This implies a clear distinction between dynamic and static scoping in tcc languages.

We believe that the results presented in this chapter contribute to a better understanding of tcc languages and to clarify some conjectures made in literature. In particular, in [SJG94a] it was conjectured that recs would be equivalent to rec1 provided that definitions are allowed to be nested within the processes. Our results show that this extension is not necessary. Another consequence of this work is that the denotational semantics of recs cannot be just an extension to sequences of the standard ccp construction in [SRP91], because the semantic equations of ccp can be satisfied only by a dynamically-scooped language.

One interesting implication of the results here presented is that, from the point of view of the expressive power, in recs the local operator is redundant. In fact, as shown in Section 8.6, recs can be encoded into rep and rep can be encoded into a local-free fragment of recs. Note that, on the contrary, locality plays a key role in the reduction of the PCP to recp and in the encoding of recp into recs.

A closely related work is [Tin99]. Also that paper explores the expressiveness of tcc languages, but it focuses on the capability of recs to encode synchronous languages. In particular, it shows that Argos [Mar92] and a version of Lustre restricted to finite domains [HCRP91] can be encoded in recs. Consequently, the decidability results extend to these synchronous languages as well.

Our notion of simulation is based on to that of [MPSS95] defined in the context of untimed ccp. In the context of tcc, [Tin01] introduced a notion of bisimulation with a complete and elegant axiomatization for the hiding-free fragment of tcc. Our notion is, however, simpler as the bisimulation of [Tin01] is based on a rather complex labeled transition system.

Except for the simulation technique, which has not yet been published, most of the results in this chapter were published as [NPV02a].
Chapter 9

Concluding Remarks

We shall conclude this dissertation with a discussion on general related work and an overall summary of its contents (more specialized related work and summaries can be found at the end of each previous chapter). We shall also describe possible directions for future research.

9.1 More on Related Work

The ntcc calculus is a strict extension of tcc [SJG94a], in the sense that tcc can be encoded in (the restricted-choice subset of) ntcc, while the vice-versa is not possible because tcc does not have constructs to express nondeterminism or unbounded finite-delay. In [SJG94a] the authors proposed also a proof system for tcc, based on an intuitionistic logic enriched with a next operator. The system, however, is complete only for the hiding-free and recursion-free fragment of tcc. In contrast our system is based on the standard classical temporal logic of [MP91] and is complete for locally independent fragment (Definition 4.3.4) which is much less restrictive (even if we confine our attention to deterministic processes) than the hiding and recursion free fragment.

An extension of tcc, which does not consider nondeterminism or unbounded finite-delay, has been proposed in [SJG96]. This extension adds strong pre-emption: the “unless” can trigger activity in the current time interval. In contrast, ntcc can only express weak pre-emption. It is argued in [dBG00], however, that in the specification of (large) timed systems weak pre-emption often suffices (and nondeterminism is crucial).

Nevertheless, strong pre-emption is important for reactive systems. In principle, strong pre-emption could be incorporated in ntcc: Semantically one would have to consider assumptions about the future evolutions of the system. As for the logic, one would have to consider a temporal extension of Default Logic [Rei80].

Other extensions of tcc have been proposed in [GJS98, GJP99]. In [GJS98] processes can evolve continuously as well as discretely. The language in [GJP99]
allows random assignments with some given distribution. None of these extensions, however, consider nondeterminism or unbounded finite-delay.

The tcgp model [DBGM00] is the only other proposal for a nondeterministic timed extension of cpp that we know of. The authors in [DBGM00] also advocate the need of nondeterminism in the context of timed cpp. In fact, they use tcgp to model interesting applications involving nondeterministic timed systems (see [DBGM00]).

One major difference between tcgp and ntcc is that in tcgp the information about the store is carried through the time units, thus the semantic setting is rather different. The notion of time is also different; in tcgp each time unit is identified with the time needed to ask and tell information to the store. As for the constructs, unlike ntcc, tcgp provides for arbitrary recursion and does not have an operator for specifying unbounded but finite delays.

A proof system for tcgp processes was recently introduced in [DBG01]. The underlying linear temporal logic in [DBG01] can be used for describing input-output behavior while our logic can only be used for the strongest-postcondition. As such the temporal logic of ntcc processes is less expressive than that one underlying the proof system of tcgp, but it is also semantically simpler and defined as the standard linear-temporal logic of [MP91]. This may come in handy when using the Consequence Rule (see Table 5.1) which is also present in [DBG01].

Timed process algebras such as [Yi91, RR88] are general formalisms for describing and analyzing timed systems. The nature of time in these algebras is rather different to that of ntcc; time, which can be discrete or dense, is associated to the individual actions of processes. In contrast, in ntcc a time unit is associated to the interval in which the system gets an input, computes on that input, and outputs the final result of the computation (hence, time is discrete). Such an ontological commitment about computation and time gives rise to a simple model with a still wide range of application, namely discrete-time reactive systems.

The relationship with linear-time temporal logics shown in this dissertation (Chapter 5), suggests that ntcc is more declarative than these process algebras, and thus it may be more suitable for the specification of discrete-time reactive systems (at least in those specifications in which a logic framework is preferred).

Functional Reactive Programming [WH00] (FRP) and Temporal Logic Programming [Mos86] (TLP) are also high-level declarative frameworks for reactive systems. In FRP programs are described in terms of continuous, time-varying, reactive values, and conditions that occur at discrete points in time. In TLP programs are described in terms of temporal formulae. The main distinction between ntcc and these frameworks arises from the underlying paradigm upon which they are defined. In ntcc the underlying paradigm is concurrent constraint programming whereas in FRP is functional programming and in TLP is logic programming. As argued in [SJG94b], in comparison with logic programming cpp provides a more algebraic view of process combinators to model concurrency. Such an algebraic view is fundamental in concurrency theory [Mil80]. FRP does not deal with algebraic issues for concurrency. In fact, only recently FRP was extended with nondeterminism to provide for parallel programming [PTS00].
9.2 Summary

In this dissertation we studied temporal concurrent constraint programming as a model for discrete-time systems by using the $\texttt{ntcc}$ calculus.

We have seen in Chapters 2 and 3 that the model is based on a few basic ideas but, as illustrated with several applications in Chapter 6, it captures interesting real world examples of discrete-time systems. We provided the calculus with a denotational semantics (Chapter 4). By identifying the exact technical problems to obtain full-abstraction, we identified a substantial fragment of the calculus for which full-abstraction holds. We defined a process linear-temporal logic with an associated inference system (Chapter 5) that can be used to express and prove that a given process satisfies a given temporal specification. Moreover, in Chapter 7 we studied decidability of the various equivalences and characterized their corresponding induced congruences. These equivalences (and their associated congruences) were proven to be decidable for a substantial fragment of the calculus. The author believes that the range of application of $\texttt{ntcc}$ and the results mentioned above suggest that the calculus is a suitable formalism for analyzing and describing discrete-time systems.

Furthermore, in Chapter 8 we established an expressive power hierarchy of several tcc languages found in the literature. These languages differ in their way of defining infinite behavior (i.e., replication or recursion) and the scope of variables (i.e., static or dynamic scope). In particular, we proved that the language with general recursion and the language with parameterless and dynamic scope are equally expressive. We also proved that the language with parameterless recursion and dynamic scope is more expressive than the language with parameterless recursion and static scope, which is in turn as expressive as the language with replication. Moreover, we also established that the less expressive languages have a decidable input-output equivalence over arbitrary constraint systems while the more expressive ones have an undecidable input-output equivalence even if we fix a finite-domain constraint system. The author believes that the tcc hierarchy contributes to a better understanding of timed ccp languages, and that the methods used for establishing the decidability results (in particular, the reduction to finite-state Büchi automata) may provide a framework to perform further systematic investigations of these languages.

9.3 Future Directions

The study presented in this dissertation is not definitive and certainly much remains to be done. The following are, in the author’s opinion, some interesting directions in the study of temporal concurrent constraint programming:

- **Decidability Issues.** Although the various process equivalences were shown to be decidable for an interesting fragment of the calculus, this dissertation leaves open the problem of the decidability of these equivalences for the full calculus. It also leaves open the decidability of the validity problem for formulae in the $\texttt{ntcc}$ process logic as well as the verification problem (i.e., whether a given process satisfies a given formula in the process logic).
• **Semantics Issues.** It would also be interesting to give a fully-abstract fixed point denotational semantics for the input-output behavior. The author believes that although $\text{ntcc}$ allows countable nondeterminism and infinite computations (see [AP86] and [Nys96] for impossibility results under these conditions), a relatively simple fully-abstract denotation may exist as a result of the particular nature of $\text{ntcc}$.

• **Extensions.** The existence of RCX program examples involving stochastic behavior, which cannot be faithfully modeled with nondeterminism, justifies the extension of the $\text{ntcc}$ to a probabilistic model. Together with Catuscia Palamidessi, the author has conducted preliminary studies using a probabilistic choice operator as done in [HP00] for the asynchronous $\pi$-calculus.

• **Application and Implementation.** Based upon the examples in Chapters 2 and the applications in Chapters 6, the author believes that the $\text{ntcc}$ calculus provides a convenient formalism for programmable micro-controllers such as RCX’s and PLC’s as well as musical applications. A rather practical research direction is then to implement $\text{ntcc}$ programming languages for these applications that benefit from the reasoning techniques provided in this dissertation. In fact, Camilo Rueda and his research group AVISPA [Rue], of which the author is also a member, are following this direction mainly in the context of computer music [RV02].

An approach for an $\text{ntcc}$-based programming of RCX and PLC micro-controllers is to compile $\text{ntcc}$ processes into finite-state automata which can then be downloaded into the controllers for execution. Recall that in Section 7.3.5 we provided, for a substantial fragment of the calculus, an algorithm to compile $\text{ntcc}$ processes into finite-state machines together with a model of execution for these machines. We were not, however, concerned with state space or performance issues. For this approach of programming micro-controllers to be successful, it is fundamental to conduct further research into these space and performance issues.
Appendix A

Proof of Correctness of the PCP Reduction

This appendix is entirely devoted to the proof of Lemma 8.3.3. We shall use the definition of the PCP \((V,W)\), and the \(\text{rec}_p\) processes \(A(index,x)\) and \(B(index,x,ok)\) all of which are given in Section 8.3.2. The lemma reads as follows:

**Lemma.** 8.3.3. \(A(index,x) \not\ni_{\circ} B(index,x,ok)\) iff there is a solution to the PCP \((V,W)\).

We shall use the following notation and some basic properties of the PCP reduction.

**Notation A.0.1.** Given a sequence \(s\), \(s^k\) denotes the sequence \(s(k),s(k + 1)\ldots\). Similarly \(\bar{s}^k\) denotes the sequence \(s(0),s(1)\ldots,s(k)\).

### A.1 Properties of the PCP Reduction

Property A.1.1 below describes the invariant of any failure-free reduction from \(A(index,x)\) to a process \(R\). In spite of its heavy notation the property can be easily explained as follows. Let us suppose that

\[
A(index,x) \xrightarrow{(c_1,d_1)} \ldots \xrightarrow{(c_m,d_m)} R
\]

where each \(d_k\), with \(1 \leq k \leq m\), is different from \(\text{false}\).

First of all, the process \(R\) must contain the local \(a's\) and \(b's\) created each time unit after the \(m\) reduction steps from \(A(index,x)\). Such variables can be represented as \(a_0, b_0,\ldots,a_m,b_m\). The process \(R\) must also specify some writing work. Let us focus on the writing of \(V\) words as the writing of \(W\) words can be explained analogously.

Since for each \(k \in \{0,\ldots,m\}\), \(d_k \neq \text{false}\) then it must be the case that \(c_k \models (index = i_k)\) for some \(i_k \in I\) (otherwise \(\text{Abort}\) would have told \(\text{false}\)).

Now, the process \(R\) must be writing some particular \(v_{i_u}\) where \(u\) is some positive number less than \(m\). Let us assume that the words

\[
v_{i_0}, \ldots, v_{i_{u-1}}
\]

147
(in this order) have been already written as well as some prefix of \( v_{i_u} \). Then 
some suffix of \( v_{i_u} \) as well the words

\[ v_{i_{u+1}}, \ldots, v_{i_m} \]

(in this order) are still to be written.

Since we write one symbol (of a \( V \) word) per time unit, it is not difficult to 
see that given \( m \), the value \( u \) mentioned above must be the least number such 
that

\[ m \leq s = \sum_{r=0}^{u} |v_r| \]

(recall that all the \( V \) words are non-empty). After a moment’s thought one can 
see that the suffix of \( v_{i_u} \) to be written should then be

\[ v_{i_u}^{[v_{i_u} \vdash (s-m)]} \]

Hence, the symbol to be written (in \( x \) at time \( m \) should be

\[ v_{i_u}^{[v_{i_u} \vdash (s-m)]} - 1(0) \]

(i.e., \( d_m = x = v_{i_u}^{[v_{i_u} \vdash (s-m)]} - 1(0)) \).

The future writing work to be done by \( R \) can be described by a process of 
the form

\[ \prod_{u < j < m} \text{wait } a_j = 1 \text{ do } V_{ij} \]

which groups those processes waiting for their \( a_{ij} \) flag (\( u < j < m \)) to be set to 
1 to start writing the corresponding \( v_{ij} \) word. The suffix of \( v_{i_u} \) to be written 
can be described by a process of the form

\[ \prod_{0 \leq k < s-m} \text{next } ^k \text{ tell}(x = v_{i_u}^{[v_{i_u} \vdash (s-m)]}(k)) \]

\[ || \text{next } ^{s-m} \text{ tell}(a_{i_u} = 0) || \text{ tell}(a_{i_{u+1}} = 1) \]

which also says that immediately after writing the suffix of \( v_{i_u} \), \( a_{i_u} = 0 \) and 
\( a_{i_{u+1}} = 1 \) announcing termination and thus allowing the writing of \( v_{i_{u+1}} \).

Furthermore, since \( c_m \vdash (index = i_m) \) for some \( i_m \in I \), the process \( R \) must 
have a call to a process \( A_{i_m}(a_m, b_m, index, x) \) in the current time interval.

We can now state the invariant property of \( A(index, x) \) as follows:

**Property A.1.1 (Safe Invariant of A).** Suppose that

\[ A(index, x) \xrightarrow{(c_1,d_1)} \ldots \xrightarrow{(c_m,d_m)} R \]

and \( d_j \neq \text{false} \) for each \( j \in \{1, \ldots, m\} \). Then there exists an index sequence 
\( i_0.i_1.\ldots.i_m \in I^* \) with \( i_0 = 0 \) s.t.

\[ R \sim_{i_0} R^m = (\text{local } a_0b_0\ldots a_mb_m) (P^m \parallel V^m \parallel Q^m \parallel W^m \parallel A_{i_m}(a_m, b_m, index, x)) \]
with

\[
P^m = \prod_{u < j < m} \text{wait } a_j = 1 \text{ do } V_{ij}
\]

\[
V^m = \prod_{0 \leq k < s - m} \text{next } k \text{tell}(x = \overrightarrow{v_{u'}}^{(s - m)(k)})
\|
\text{next } s - m(\text{tell}(a_u = 0) \text{ or } \text{tell}(a_{u+1} = 1))
\]

\[
Q^m = \prod_{u' < j < m} \text{wait } b_j = 1 \text{ do } W_{ij}
\]

\[
W^m = \prod_{0 \leq k < s' - m} \text{next } k \text{tell}(x = \overrightarrow{w_{u'}}^{(s' - m)(k)})
\|
\text{next } s' - m(\text{tell}(b_{u'} = 0) \text{ or } \text{tell}(b_{u'+1} = 1))
\]

\[
d_m \models x = \overrightarrow{v_{u'}}^{(|v_{u'}| - (s - m)) - 1}(0) \land x = \overrightarrow{w_{u'}}^{(|w_{u'}| - (s' - m)) - 1}(0)
\]

where \( u, u' \) are the least numbers satisfying,

\[
m \leq s = \sum_{r=0}^{u} |v_r| \text{ and } m \leq s' = \sum_{r=0}^{u'} |w_r|.
\]

**Proof.** The proof proceeds by induction on \( m \). Suppose that \( A(index, x) \xrightarrow{(c_1, d_1)} \ldots \xrightarrow{(c_m, d_m)} R \) and \( d_j \neq \text{false} \) for each \( j \in \{1, \ldots, m\} \). For simplicity, we shall omit the parameters of \( A \).

Case \( m = 1 \). Since \( d_1 \neq \text{false} \) then it must be the case that \( c_i \models (\text{index} = i_1) \) for some \( i_1 \in I \) (otherwise \text{Abort} would have told \text{false}). One can verify that

\[
R \equiv (\text{local } a \ b \ a' \ b' \ \text{ichosen}) (\prod_{0 \leq k < |v_0| - 1} \text{next } k \text{tell}(x = \overrightarrow{v_0}^k(0))
\|
\text{next } |v_0| - 1(\text{tell}(a = 0) \text{ or } \text{tell}(a' = 1))
\|
\prod_{0 \leq k < |w_0| - 1} \text{next } k \text{tell}(x = \overrightarrow{w_0}^k(0))
\|
\text{next } |w_0| - 1(\text{tell}(b = 0) \text{ or } \text{tell}(b' = 1))
\|
A_{i_1}(a', b', \text{index}, x))
\]

and \( d_1 \models x = v_0(0) \land x = w_0(0) \). One can remove the local variable \( \text{ichosen} \) and rename \( a, b, a' \) and \( b' \) as \( a_0, b_0, a_1 \) and \( b_1 \) (resp.) in the process \( R \) without changing the input-output behavior. Then by taking \( u = u' = 0 \) (and thus \( s = |v_0| \) and \( s' = |w_0| \)) we can prove that \( R \sim_i R^1 = (\text{local } a_0 b_0 a_1 b_1)(P^1 \parallel V^1 \parallel Q^1 \parallel W^1 \parallel A_{i_1}(a_1, b_1, \text{index}, x)) \).

Case \( m > 1 \). Let \( R' \) s.t., \( A \xrightarrow{(c_1, d_1)} \ldots \xrightarrow{(c_{m-1}, d_{m-1})} R' \). By appealing to induction we know that there exists \( i_0, \ldots, i_{m-1} \in I^* \) with \( i_0 = 0 \) s.t.,

\[
A^{m-1} = (\text{local } a_0 b_0 \ldots a_{m-1} b_{m-1})(P^{m-1} \parallel V^{m-1} \parallel Q^{m-1} \parallel W^{m-1}
\|
A_{i_{m-1}}(a_{m-1}, b_{m-1}, \text{index}, x))
\]
is input-output equivalent to \( R' \). Since the \( \text{rec}_p \) is deterministic, it suffices to show that for some \( R'' \), \( A^{m-1} \xrightarrow{(c_m,d_m)} R'' \) and \( R'' \sim io A^m \).

Let \( u_{m-1} \) and \( u'_{m-1} \) be the least numbers s.t.,

\[
m - 1 \leq s_{m-1} = \sum_{r=0}^{u_{m-1}} |v_r| \quad \text{and} \quad m - 1 \leq s'_{m-1} = \sum_{r=0}^{u'_{m-1}} |w_r|.
\]

Intuitively, \( u_{m-1} \) and \( u'_{m-1} \) are the \( u \) and \( u' \) in \( A^{m-1} \).

We need to consider various cases depending whether \( m - 1 = s_{m-1} \) or \( m - 1 < s_{m-1} \), and whether \( m - 1 = s'_{m-1} \) or \( m - 1 < s'_{m-1} \). Let us confine our attention to the case \( m - 1 = s_{m-1} \) and \( m - 1 < s'_{m-1} \), the other cases are similar.

Since \( d_m \neq \text{false} \) then it must the case that \( c_m \models (\text{index} = i_m) \) for some \( i_m \in I \). Then, it follows that \( A^{m-1} \xrightarrow{(c_m,d_m)} R'' \) where

\[
R'' \equiv (\text{local} a_1 b_1 \ldots a_{m-1} b_{m-1})(
\prod_{u_{m-1}+1 < j < m-1} \text{wait } a_j = 1 \text{ do } V_{ij}
\prod_{u'_{m-1} < j < m-1} \text{wait } b_j = 1 \text{ do } W_{ij}
\prod_{0 \leq k < s'_{m-1} - m} \text{next } k \text{ tell}(x = \overrightarrow{u_{m-1}} | v_{u_{m-1}} | -(s'_{m-1} - m)(k))
\text{next } s'_{m-1} - m(\text{tell}(b_{u_{m-1}} = 0) \parallel \text{tell}(b_{u_{m-1}+1} = 1))
(\text{local} a' \ b' \ ichosen)(
\text{wait } a' = 1 \text{ do } V_{im}
\text{wait } b' = 1 \text{ do } W_{im}
\prod_{0 \leq k < |v_{u_{m-1}+1}|-1} \text{next } k \text{ tell}(x = \overrightarrow{u_{m-1}+1} | v_{u_{m-1}+1})(k))
\text{next } |v_{u_{m-1}+1}|-1(\text{tell}(a_{u_{m-1}} = 0) \parallel \text{tell}(a_{u_{m-1}+1} = 1))
A_{im}(a',b',\text{index},x))
\]

and \( d_m \models x = v_{u_{m-1}+1}(0) \land x = \overrightarrow{u_{m-1}} | v_{u_{m-1}} | -(b'_{m-1} - m) - 1(0) \)

In the process above one can remove \( ichosen \) and rename \( a' \) and \( b' \) as \( a_m \) and \( b_m \) (resp.) and then move them to the outermost position without changing its input-output behavior. Then, by taking \( u = u_{m-1} + 1 \) and \( u' = u'_{m-1} \) and making some simple rearrangements of the products we can verify that \( R'' \sim io R^m \) as wanted.

\[ \square \]

Below we give the invariant property of process \( B(\text{index}, x, ok) \). The invariant looks exactly as that of \( A(\text{index}, x) \) in Property A.1.1, except for the process

\[
\prod_{0 \leq j < m} \text{ whenever } a_{ij} = 0 \land b_{ij} = 0 \text{ do } \text{tell}(ok = 1)
\]

which tells \( ok = 1 \) whenever the writing of two words \( v_{ij} \) and \( w_{ij} \) finishes in the same time interval.
Property A.1.2 (Safe Invariant of $B$). Suppose that

$$B(index, x, ok) \xrightarrow{(c_1,d_1)} \cdots \xrightarrow{(c_m,d_m)} B'$$

and $d_j \neq \text{false}$ for each $j \in \{0,1,\ldots,m\}$. Then, there exists an index sequence $i_0,i_1,\ldots,i_m \in I^*$ with $i_0 = 0$ s.t.,

$$B' \sim_{i_0} (\text{local } a_0 a_1 a_m b_m) (**m** || **V** || **Q** || **W** || $B_{i_0} (a_{i_0}, b_{i_0}, index, x, ok)$ \prod_{0 \leq j < m \text{ whenever } a_{i_j} = 0 \land b_{i_j} = 0 \text{ do tell}(ok = 1))$$

with $**m**, **V**, **Q**, and **W** as in Property A.1.1.

Proof. The proof proceeds by induction on $n$ very much the same way as the proof of Property A.1.1. \hfill \square

The following property states the kind of input-output pairs of sequences distinguishing $A(index, x)$ from $B(index, x, ok)$.

Property A.1.3 (Distinguishing IO Sequences).

$$A(index, x) \not\sim_{i_0} B(index, x, ok)$$

iff for some $(\alpha,\alpha') \in \text{io}(B(index, x, ok))$, there exists $m$ s.t.,

1. $\alpha(m) \not\models (ok = 1)$, $\alpha'(m) \models (ok = 1)$,

2. $\alpha'(m) \not\models \text{false}$,

3. for all $m' < m$, $\alpha'(m') \not\models (ok = 1)$.

Proof. The “if” direction is trivial; $A$ cannot tell $(ok = 1)$. Consider the “only if” direction. Because of the process whenever $a = 0 \land b = 0$ do tell$(ok = 1)$, it is not difficult to see that there must be a $(\alpha,\alpha') \in \text{io}(B)$ such that there exists $m$, satisfying Items (1-2). Take the first such an $m$. Therefore, for all $m' < m$, $\alpha'(m') \not\models (ok = 1)$ but provided that $\alpha'(m') \not\models \text{false}$. However, if for some $m' < m$, $\alpha'(m')$ were false then, by looking at the behavior of process $\text{Abort}$, for each $m'' > m'$, $\alpha'(m'')$ would be false as well. This would imply that $\alpha'(m) \models \text{false}$, thus contradicting Item 2. Hence, Item 3 is also satisfied. \hfill \square

A.2 Proof of Lemma 8.3.3

We now turn to proof of the main lemma of this appendix: Lemma 8.3.3. We want to prove that $A(index, x) \not\sim_{i_0} B(index, x, ok)$ if and only if there is a solution to the PCP $(V,W)$.

Proof. For simplicity we abbreviate $A(index, x)$ and $B(index, x, ok)$ as $A$ and $B$, respectively.

Let us consider the “only if” direction. Suppose that $A \not\sim_{i_0} B$. Let $(\alpha,\alpha') \in \text{io}(B)$, with $\alpha' = d_1 d_2 \ldots$, and $m$ satisfying Items (1-3) in Property A.1.3.
From Property A.1.2 it follows that there exist \( i_1, i_2, \ldots, i_u \in I \), such that \( d_k \models x = v(k) \) (0 \( \leq k < |v| = m + 1 \)) where

\[
v = v_0.v_{i_1} \ldots \hat{v}_{i_u}^{-p}
\]  

with \( p = n - \sum_{r=0}^{u-1} |v_r| \). Intuitively, \( v \) denotes the \( V \) sequence that have been written by \( B \) after \( m + 1 \) steps.

Similarly, for the same choice of \( i_1, i_2, \ldots, i_u \in I \), \( d_k \models x = w(k) \) (0 \( \leq k \leq |w| \)) where

\[
w = w_0.w_{i_1} \ldots \hat{w}_{i_u}^{-q}
\]  

with \( q = n - \sum_{r=0}^{u'-1} |w_{i_r}| \).

With the help of A.1.2 we prove that, since \( d_m \models (ok = 1) \), for some \( j < m \), two local variables \( a_{i_j} \) and \( b_{i_j} \) were set to 0 in the \( m \)-th time unit. Also, notice that each \( a_k \) and \( b_k \) \( (0 \leq k \leq m) \) can be set to 0 only once. Furthermore, if \( k < u \) (resp., \( k < u' \)) then \( a_{i_k} \) (resp., \( b_{i_k} \)) must have been set to 0 in some \( m' \)-th time unit before the \( m \)-th one and if \( k > u \) (resp., \( k > u' \)) then \( a_{i_j} \) (resp., \( b_{i_k} \)) must not have been set to 0. It follows that \( j = u = u' \).

Now, the variable \( a_{i_u} \) is set to 0 only when the last prefix \( v_{i_u} \) has been written – this follows from the definition of \( V^m \) in Property A.1.2. Therefore, \( \hat{v}_{i_u}^{-p} = v_{i_u} \). Similarly, we conclude that \( \hat{w}_{i_u}^{-q} = w_{i_u} \). Hence, \( v = v_0.v_{i_1} \ldots v_{i_u} \) and \( w = w_0.w_{i_1} \ldots w_{i_u} \), from Equations A.1 and A.2.

Finally, since \( d_k \neq \text{false} \) and \( d_i \models x = v(k) \land x = w(k) \) for 0 \( \leq k \leq u \) then \( v_0.v_{i_1} \ldots v_{i_u} = w_0.w_{i_1} \ldots w_{i_u} \). This implies that there is a solution to the PCP \((V, W)\).

As for the “if” direction suppose that \( i_1, \ldots, i_m \) is a solution to the PCP \((V, W)\). By taking \( \alpha = (\text{index} = i_1) \ldots (\text{index} = i_m).\text{true}^u \), one can verify that \( (\alpha, \alpha') \in io(B) \) with \( \alpha, \alpha' \) and \( m \) satisfying Items (1-3) in Property A.1.3. Therefore, \( B \not\models_{io} A \). \( \square \)
Appendix B

Proof of Correctness of the Encodings

This appendix is devoted to the proof of Theorem 8.7.1 which states the correctness of the various tcc encodings wrt to the input-output equivalence.

Theorem. 8.7.1 For every encoding $[\cdot] : \ell \rightarrow \ell$ defined from Section 8.6.1 through Section 8.6.6, we have $io(P) = io([P])$.

The main proof technique we use in this appendix is that of simulation introduced in Section 8.7. The idea is that to prove input-output equivalence between $P$ and $Q$ (possibly in different tcc languages), it suffices to exhibit two simulations: One including $((P, \text{true}), (Q, \text{true}))$ and the other including $((Q, \text{true}), (P, \text{true}))$. The reader may also need to have a look at the notations introduced in Convention 8.1.4 and Notation 8.6.1 and 8.7.2.

In the following section we shall prove, one by one, the correctness of the various encodings. The order in which we shall present the proofs does not correspond to the order the encodings were introduced in Section 8.6 but rather to how similar their proofs are.

We shall use the syntactic notion of the size of a process.

Definition B.0.1. Let $P$ be either a rec or a rep process. The size of $P$, written $|P|$, is recursively given by:

\[
\begin{align*}
|\text{skip}| &= |\text{tell}(c)| = |A(\overline{x})| = 1 \\
|P \parallel Q| &= 1 + |P| + |Q| \\
|!P| &= |\text{next } P| = |\text{unless } c \text{ next } P| = |\text{when } c \text{ do } P| = |(\text{local } x) \ P| \\
&= 1 + |P|
\end{align*}
\]

B.1 Correctness of the encoding $\text{rep} \rightarrow \text{rec}_1$

The following lemma provides the simulations witnessing the similarity between a process in $\text{rep}$ and its encoding in $\text{rec}_1$.

Lemma B.1.1. Let $[\cdot] : \text{rep} \rightarrow \text{rec}_1$ be the encoding in Definition 8.6.4. Let $B$ be the relation

\[
\{( (P, c), ([P], c) ) \mid P \text{ is in } \text{rep} \}.
\]

The relations $B$ and $B^{-1}$ are simulations.

153
Proof. We need prove that $B$ and $B^{-1}$ satisfy the Condition 1-3 in Definition 8.7.7. Conditions 1 and 3 can be easily established. Let us focus our attention on Condition 2.

**Condition 2 for $B$.** Let $(\gamma_1, \gamma'_1) \in B$ and suppose that $\gamma_1 \rightarrow \gamma_2$. We shall exhibit a $\gamma'_2$ s.t., $\gamma'_1 \rightarrow \gamma'_2$ with $\gamma_2 B \gamma'_2$.

The proof proceeds by induction on the depth of the inference of $\gamma_1 \rightarrow \gamma_2$ considering the possible cases of the last step of inference. We content ourselves with illustrating the key case: REP. In this case, $\gamma_1 = \langle [P], c \rangle$ and $\gamma_2 = \langle P \parallel \text{next}!P, c \rangle$. By using REC, $\gamma'_1 = \langle [[P]], c \rangle = \langle R_P(\bar{x}), c \rangle \rightarrow \gamma'_2 = \langle [[P]] \parallel \text{next} R_P(\bar{x}), c \rangle$. From the definition of $[.]$ it is easy to see that $\gamma'_2 B \gamma'_2$ as desired.

**Condition 2 for $B^{-1}$.** Let $(\gamma'_1, \gamma_1) \in B^{-1}$ and suppose that $\gamma'_1 \rightarrow \gamma'_2$. We shall exhibit $\gamma_2$ s.t., $\gamma_1 \rightarrow \gamma_2$ with $\gamma_2 B^{-1} \gamma_2$.

The proof proceeds by induction on the depth of the inference of $\gamma'_1 \rightarrow \gamma'_2$ considering the possible cases of the last step of inference. Here we confine ourselves to the key case: REC. In this case, $\gamma'_1 = \langle [[P]], c \rangle = \langle R_P(\bar{x}), c \rangle$ and $\gamma_2 = \langle [P] \parallel \text{next} R_P(\bar{x}), c \rangle$. By using REP, $\gamma_1 = \langle [P], c \rangle \rightarrow \gamma_2 = \langle P \parallel \text{next}!P, c \rangle$. From the definition of $[.]$ it is easy to see that $\gamma'_2 B \gamma_2$ as wanted.

The correctness of the encoding is an immediate consequence of the above lemma.

**Theorem B.1.2 (Correctness of $[.] : \text{rep} \rightarrow \text{rec}_1$).** Let $[.] : \text{rep} \rightarrow \text{rec}_1$ be the encoding in Definition 8.6.4. The processes $P$ and $[P]$ are input-output equivalent.

**Proof.** It follows from Lemma B.1.1 and Theorem 8.7.13. \[\square\]

### B.2 Correctness of the encoding $\text{rec}_d \rightarrow \text{rec}_p$

The correctness of this encoding can be established along the lines of the previous proof of correctness.

We start by exhibiting the simulations witnessing similarity.

**Lemma B.2.1.** Let $[.] : \text{rec}_d \rightarrow \text{rec}_p$ be the encoding in Definition 8.6.5. Let $B$ be the relation

$$\{(\langle P, c \rangle, \langle [[P]], c \rangle) \mid P \text{ is in } \text{rec}_d \}.$$  

The relations $B$ and $B^{-1}$ are simulations.

**Proof.** We need prove that $B$ and $B^{-1}$ satisfy the Condition 1-3 in Definition 8.7.7. Condition 1 and 3 can be easily established. Let us focus our attention on Condition 2.
Condition 2 for $B$ Let $(\gamma_1, \gamma'_1) \in B$ and suppose that $\gamma_1 \rightarrow \gamma_2$. We shall identify a $\gamma'_2$, s.t. $\gamma'_1 \rightarrow \gamma'_2$ with $\gamma_2 B \gamma'_2$.

The proof proceeds by induction on the depth of the inference of $\gamma_1 \rightarrow \gamma_2$ considering the possible cases of the last step of inference. Let us illustrate the key case: REC.

So, $\gamma_1 = \langle A, c \rangle$ and $\gamma_2 = \langle P, c \rangle$ with $A \overset{\text{def}}{=} P \in D_{\text{rec}_d}$. We have $A(\vec{x}) \overset{\text{def}}{=} P \in D_{\text{rec}_r}$. Thus, by using REC, $\gamma'_1 = \langle [A], c \rangle = \langle A(\vec{x}), c \rangle \rightarrow \gamma'_2 = \langle [P] [x / \vec{x}], c \rangle = \langle [P], c \rangle$.

From the definition of $[\cdot]$, we observe that $\gamma_2 B \gamma'_2$ as desired.

Condition 2 for $B^{-1}$ Similar to the previous case.

The correctness of the encoding follows from the lemma above.

**Theorem B.2.2 (Correctness of $[\cdot] : \text{rec}_d \rightarrow \text{rec}_p$).** Let $[\cdot] : \text{rec}_d \rightarrow \text{rec}_p$ be the encoding in Definition 8.6.5. The processes $P$ and $[P]$ are input-output equivalent.

**Proof.** It follows from Lemma B.2.1 and Theorem 8.7.13.

**B.3 Correctness of $[\cdot] : \text{rec}_s \rightarrow \text{rep}$**

The proof of correctness of this encoding is somewhat more complex due to the way recursion and locality are encoded.

We shall use the following properties of the encoding.

**Lemma B.3.1.** Let $[\cdot] : \text{rec}_s \rightarrow \text{rep}$ be the encoding in Definition 8.6.2:

1. $\langle [A], c \rangle \sim_w \langle [P], c \rangle$ where $A \overset{\text{def}}{=} P$.

2. $\langle ([\text{local} x] P), c \rangle \sim \langle ([\text{local} y] [P[y/x]], c) \rangle$ where $y$ is fresh.

**Proof.** 1 Notice that $[A]$ takes the form

$$P_1 = (\text{local } \vec{z}) \langle \text{tell}(\text{call}(z_A)) \parallel (\text{!when call}(z_A) \text{ do } [P]) \parallel R \rangle$$

and $[P]$ takes the form

$$Q_1 = (\text{local } \vec{z}) \langle [P] \parallel (\text{!when call}(z_A) \text{ do } [P]) \parallel R \rangle$$

where $R = \prod_{i=2}^{n} \text{!when call}(z_{A_i}) \text{ do } [P_i]_0$ and $\vec{z} = z_A, z_{A_1}, \ldots, z_{A_n}$.

We now exhibit the simulations witnessing the similarity of $\langle [A], c \rangle$ and $\langle [P], c \rangle$. For the simplicity of the presentation, we assume that $|\vec{z}| = 1$.

Let $B \subset \Gamma_{\text{rep}} \times \Gamma_{\text{rep}}$ be defined as

$$\{ \langle \langle P^d_i, c \rangle, \langle Q^d_j, c \rangle \rangle | 1 \leq i \leq 5, 1 \leq j \leq 2 \}$$
where for $T = \text{when } \text{call}(z_A) \text{ do } [P]$

\[
\begin{align*}
P^d_1 &= (\text{local } \tilde{z}, d)(\text{tell}(\text{call}(z_A)) \parallel !T \parallel R) \\
P^d_2 &= (\text{local } \tilde{z}, d \wedge \text{call}(z_A))(!T \parallel R) \\
P^d_3 &= (\text{local } \tilde{z}, d)(\text{tell}(\text{call}(z_A)) \parallel (T \parallel \text{next }!T) \parallel R) \\
P^d_4 &= (\text{local } \tilde{z}, d \wedge \text{call}(z_A))((T \parallel \text{next }!T) \parallel R) \\
P^d_5 &= (\text{local } \tilde{z}, d \wedge \text{call}(z_A))(((P) \parallel \text{next }!T) \parallel R) \\
Q^d_1 &= (\text{local } \tilde{z}, d)([[P]] \parallel T \parallel R) \\
Q^d_2 &= (\text{local } \tilde{z}, d)([[P]] \parallel T \parallel \text{next }!T \parallel R)
\end{align*}
\]

We can verify that $B$ and $B^{-1}$ are simulations. Therefore, Item 1 follows from the fact that $\langle[A], c \rangle, \langle[P], c \rangle \in B$ and $\langle[[P]], c \rangle, \langle[A], c \rangle \in B^{-1}$.

It is not difficult to prove that $B^{-1}$ is a simulation. As for $B$, the only problem is when we need to execute $[P]$ in an internal evolution of $P_1$, e.g., $P_5$, to match the execution of the $[P]$ in $Q_1$. Notice that $!T$ is next-guarded in $P_5$, but not in $Q_1$. Nevertheless, from our requirements about recursion being next guarded (see Section 8.1.2), we conclude that $\text{call}(z_A)$ will not be told during the current time interval by any evolution of $Q_1$ — i.e., $T$ in $Q_1$ or $Q_2$ will not trigger its $[[P]]$ in the current time interval.

2 Notice that $[[\text{local } x] P]$ takes the form

\[
P_1 = (\text{local } \tilde{z})((\text{local } y)[P[y/x]]_0) \parallel R)
\]

and $([\text{local } x] P[y/x])$ takes the form

\[
Q_1 = (\text{local } y)(\text{local } \tilde{z})([[P[y/x]]_0) \parallel R)
\]

where $y$ is fresh.

Because $y$ is fresh, it is different from each variable in $\tilde{z}$ and it does not occur in $R$. Hence, $y$ can be "pulled" into the outermost position of $P_1$ to get $Q_1$. This follows from the more general properties: For any two processes $Q, Q'$, $\langle(\text{local } x)(\text{local } y) Q, \text{true} \rangle \sim \langle(\text{local } y)(\text{local } x) Q, \text{true} \rangle$, and $\langle(\text{local } x) Q \parallel Q', \text{true} \rangle \sim \langle(\text{local } x) Q \parallel Q', \text{true} \rangle$, if $x \notin \text{fv}(Q)$. The proof of such properties is standard.

\[\square\]

Notice that in the simulation below the simulating configuration is quotiented by weak similarity.

**Lemma B.3.2 ([P] simulates P).** Let $[: \text{rec} \rightarrow \text{rep}]$ be the encoding in Definition 8.6.2. Let $B$ be the relation

\[
\{((P, c), [[P]], [c]) \mid P \text{ is in rec} \}
\]

The relation $B \sim_w$ is a simulation.

**Proof.** It is sufficient to prove that $B \sim_w$ satisfies Items 1-3 in the condition of Proposition 8.7.11. Item 1 holds trivially. Let us consider Items 2-3.
2 We need to show that for each \((P, c), (P', c)\) \in B, if \(\gamma_1 = (P, c) \rightarrow \gamma_2 = (Q, d)\) then there is \(\gamma'_2\) s.t., \(\gamma'_1 = (P', c) \rightarrow^* \gamma'_2\) and \(\gamma_2 B \sim_w \gamma'_2\).

Suppose that \(\gamma_1 = (P, c) \rightarrow \gamma_2 = (Q, d)\). We shall exhibit a \(\gamma'_2\) s.t., \(\gamma'_1 = (P', c) \rightarrow^* \gamma'_2\) and \(\gamma_2 B \sim_w \gamma'_2\). We proceed by induction (on the depth) of the inference of \(\gamma_1 \rightarrow \gamma_2\). The key cases are the local operator and the identifiers.

**Using REC** Then \(\gamma_1 = (A, c)\) and \(\gamma_2 = (Q, c)\) where \(A \overset{\text{def}}{=} Q\). From Lemma B.3.1.1, \(\langle [A], c \rangle \sim_w \langle [Q], c \rangle\), thus \(\langle [A], c \rangle \rightarrow^0 \gamma'_2 \sim_w \langle [Q], c \rangle\) as wanted.

**Using LOC'** In this case we have \(\gamma_1 = (\text{local}\ x, P, c)\) and \(\gamma_2 = (Q, d)\) with \(\langle P[y/x], c \rangle \rightarrow (Q, d)\), \(y\) being fresh, by a shorter inference. By appeal to induction, we let \(\gamma' = (R, c \land e)\) be such that \(\langle P[y/x], c \rangle \rightarrow^* \gamma'\) and \(\gamma_2 B \sim_w \gamma'\). Therefore,

\[
\langle R, c \land e \rangle \sim_w \langle [Q], d \rangle \tag{B.1}
\]

Notice that \(y\) is fresh and it does not occur in \(c\), so \(\exists y c = c\). Since \(\langle P[y/x], c \rangle \rightarrow^* \gamma'\), we can use LOC to obtain

\[
\langle (\text{local}\ y)[P[y/x]], c \rangle \rightarrow^* \langle (\text{local}\ y, c \land e) R, c \land \exists y e \rangle \tag{B.2}
\]

Using the fact that \(y\) is a fresh variable, one can show that \(\langle (\text{local}\ y, c \land e) R, c \land \exists y e \rangle \sim_w \langle R, c \land e \rangle\). Hence, from Equation B.1,

\[
\langle (\text{local}\ y, c \land e) R, c \land \exists y e \rangle \sim_w \langle [Q], d \rangle \tag{B.3}
\]

Furthermore, from Lemma B.3.1.2 (and Proposition 8.7.10), we have \(\langle [\text{local}\ x) P], c \rangle \sim_w \langle [\text{local}\ y] P[y/x], c \rangle\), thus from Equations B.2 and B.3, we obtain \(\langle [\text{local}\ x) P], c \rangle \rightarrow^* \gamma'_2 \sim_w \langle [Q], d \rangle\), as wanted.

3 We need to show that for each \((P, c), (P', c)\) \in B, if \(\gamma_1 = (P, c) \not\rightarrow\) then we can exhibit a \(\gamma'_2\) s.t., \(\gamma'_1 = (P', c) \rightarrow^* \gamma'_2 = (Q', c) \not\rightarrow\) and furthermore, \((F(P), \text{true}) B \sim_w (F(Q'), \text{true})\).

We proceed by induction on the size of \(P, |P|\). Consider the case \(|P| = 1\). If \(P = \text{skip}\), \([P] = \text{skip}\) so we are done, take \(\gamma'_2 = \gamma_1\). If \(P = A\) or \(P = \text{tell}(e)\) then the property holds trivially since \((P, c)\) has one transition. As for the case \(|P| > 1\), the key case is the local operator.

**Case** \(P = (\text{local}\ x) Q\). We can infer that \((Q[y/x], c) \not\rightarrow\) for an arbitrary fresh variable \(y\). Notice \(|Q[y/x]| < |P|\), thus by appeal to induction we conclude that there must exist \((R, c)\) s.t.,

\[
\langle [Q[y/x]], c \rangle \rightarrow^* \langle R, c \rangle \not\rightarrow \tag{B.4}
\]

and \((F(Q[y/x]), \text{true}) B \sim_w (F(R), \text{true})\). Therefore,

\[
(F(R), \text{true}) \sim_w (F(Q[y/x]), \text{true}) \tag{B.5}
\]
From the definition of $F$ (Definition 3.3.10), it must follow that 
$\langle [F(Q[y/x]), \text{true}] \rangle = \langle [F(Q)[y/x]], \text{true} \rangle$. Hence,

$$\langle F(R), \text{true} \rangle \sim_w \langle [F(Q)[y/x]], \text{true} \rangle \quad (B.6)$$

With the help of LOC and using the reduction sequence in Equation B.4, and the fact $\exists y c = c$ (as $\gamma$ is fresh, it does not occur in $c$) we obtain

$$\langle (\text{local}) [Q[y/x]], c \rangle \rightarrow^* \langle (\text{local}) y, c \rangle R \rightarrow c \quad (B.7)$$

But from Lemma B.3.1(2) (and Proposition 8.7.10) we must have 
$\langle [P], c \rangle \sim_w \langle (\text{local}) [Q[y/x]], c \rangle$. So, from Equation B.7,

$$\langle [P], c \rangle \rightarrow^* \sim_w \langle (\text{local}) y, c \rangle R \rightarrow c .$$

Take $\gamma'_2 = \langle (\text{local}) y, c \rangle R \rightarrow c$. We can use the definition of $F$, and then Equation B.6 together with the fact that $\gamma$ is a fresh variable, to derive

$$\langle F((\text{local}) y, c) R, \text{true} \rangle = \langle (\text{local}) y F(R), \text{true} \rangle$$

$$\sim_w \langle (\text{local}) F([Q[y/x]], \text{true} \rangle$$

But $\langle (\text{local}) [F(Q)[y/x]], \text{true} \rangle \sim_w \langle (\text{local}) F(Q), \text{true} \rangle$ by using Lemma B.3.1(2) (and Proposition 8.7.10). Since from the definition of $F$, $F(P) = (\text{local}) x F(Q)$, we are done.

$\square$

In the simulation below the simulated configuration is quotiented by weak similarity.

**Lemma B.3.3 (P simulates $\llbracket P \rrbracket$).** Let $[\cdot] : \text{rec} \rightarrow \text{rep}$ be the encoding in Definition 8.6.2. Let $B$ be the relation

$$\{ \langle (P, c), \llbracket P \rrbracket, c \rangle \mid P \text{ is in } \text{rec} \}.$$  

The relation $\sim_w B^{-1}$ is a simulation.

**Proof.** It is sufficient to prove that $\sim_w B^{-1}$ satisfies Items 1-3 in the condition of Proposition 8.7.12. Item 1 holds trivially. Item 3 can be established along the lines of the proof of Item 3 in Lemma B.3.2. Here we confine ourselves to Item 2.

2 We shall show that for all $n$, for each $(\gamma'_1, \gamma_1) = \langle \llbracket P \rrbracket, c_1 \rangle$, $(P, c_1) \in B^{-1}$, if $\gamma'_1 \rightarrow^n \gamma'_2$ then we can find a $\gamma_2$ s.t. $\gamma_1 \rightarrow^* \gamma_2$ and $\gamma_2 \sim_w B^{-1} \gamma_2$.

Suppose that $\gamma'_1 \rightarrow^n \gamma'_2$. We shall exhibit a $\gamma_2$ such that $\gamma_1 \rightarrow^* \gamma_2$ and $\gamma_2 \sim_w B^{-1} \gamma_2$. The proof proceeds by induction on $n$ and on the size of the process in $\gamma'_1$. We shall illustrate the key cases: the encoding of the identifiers and the local operators.
Identifiers Assume $\gamma_1 = \langle [A], c_1 \rangle$ with $A \overset{\text{def}}{=} R$. Suppose that $n \leq 2$. One can verify that $\langle [R], c_1 \rangle \sim_w \gamma_2$. Since $\gamma_1 = \langle A, c_1 \rangle \rightarrow \langle R, c_1 \rangle$, we are done; take $\gamma_2 = \langle R, c_1 \rangle$.

Suppose $n > 2$. From the encoding of $[A]$, one can see that if $\gamma_1 \rightarrow^n \gamma_2$ and $n > 2$ then $\langle [R], c_1 \rangle \rightarrow^m \sim_w \gamma_2$ with $m < n$.

Since $m < n$, we use the induction to obtain $\langle R, c_1 \rangle \rightarrow^* \gamma_2$, with $\gamma_2 \sim B^{-1} \gamma_2$ as desired (notice that $\langle A, c_1 \rangle \rightarrow \langle R, c_1 \rangle$ by REC).

Local Operators Let us assume that $\gamma_1 = \langle [(\text{local } P)], c_1 \rangle$. From Lemma B.3.1(2), $\gamma_1$ is strongly similar to $\langle (\text{local } y) [P[y/x]], c_1 \rangle$ with $y$ being fresh. Hence, from the operational semantics we infer that there must be a sequence

\[
\langle (\text{local } y) [P[y/x]], c_1 \rangle \rightarrow^n \langle (\text{local } y, c_2) R_2, c_2 \rangle \sim \gamma_2
\]

with $\langle [P[y/x]], c_1 \rangle \rightarrow^n \langle R_2, c_2 \rangle$.

Now, the size of $[P[y/x]]$ is less than the size of $[(\text{local } x) P]$. Hence, since $\langle [P[y/x]], c_1 \rangle \rightarrow^n \langle R_2, c_2 \rangle$, we use the induction to conclude that for some configuration $\langle Q, d \rangle$, we must have $\langle P[y]/y, c_1 \rangle \rightarrow^* \langle Q, d \rangle$ and $\langle R_2, c_2 \rangle \sim B^{-1} \langle Q, d \rangle$. Therefore, $\langle R_2, c_2 \rangle \sim_w \langle [Q], d \rangle$.

Furthermore, using the fact that $y$ is fresh, we can verify that $\langle (\text{local } y, c_2) R_2, c_2 \rangle \sim_w \langle R_2, c_2 \rangle$.

So, in summary we have

$\gamma_1 \rightarrow^n \gamma_2 \sim_w \langle (\text{local } y, c_2) R_2 \rangle \sim_w \langle R_2, c_2 \rangle \sim_w \langle [Q], d \rangle$.

Since $\langle P[x/y], c_1 \rangle \rightarrow^* \langle Q, d \rangle$, we can use the rule LOC to obtain $\gamma_1 = \langle (\text{local } x) P, c_1 \rangle \rightarrow^* \langle Q, d \rangle = \gamma_2$. Therefore, $\gamma_2 \sim_w B^{-1} \gamma_2$, as wanted.

\[\square\]

From the lemmas above, we obtain the correctness of the encoding.

**Theorem B.3.4 (Correctness of $\llbracket \cdot \rrbracket : \text{rec}_s \rightarrow \text{rep}$).** Let $\llbracket \cdot \rrbracket : \text{rec}_s \rightarrow \text{rep}$ be the encoding in Definition 8.6.2. The processes $P$ and $[P]$ are input-output equivalent.

**Proof.** Immediate from Lemmas B.3.2 and B.3.3, and Theorem 8.7.13. \[\square\]

### B.4 Correctness of $\llbracket \cdot \rrbracket : \text{rec}_1 \rightarrow \text{rep}$

The proof of correctness of the encoding from $\text{rec}_1$ to $\text{rep}$ proceeds along the lines of that of the previous section. The basic difference is that instead of using Lemma B.3.1(1) we use the following version of it.

**Lemma B.4.1.** Let $\llbracket \cdot \rrbracket : \text{rec}_1 \rightarrow \text{rep}$ be the encoding in Definition 8.6.3: $\langle [A(y)], c \rangle \sim_w \langle [P[y/x]], c \rangle$ where $A(\bar{x}) \overset{\text{def}}{=} P$. 
Proof. Similar to the proof of Lemma B.3.1(1). □

The simulations are given by the following lemma whose proof proceeds as the proof of Lemma B.3.2 (and Lemma B.3.3).

**Lemma B.4.2.** Let \([\cdot] : \text{rec}_1 \rightarrow \text{rep}\) be the encoding in Definition 8.6.3. Let \(B\) be the relation
\[
\{ (\langle P, c \rangle, \langle [P], c \rangle) \mid P \text{ is in } \text{rec}_1 \}\}
\]
The relations \(\sim_w B\) and \(\sim_w B\) are simulations.

Proof. The proof proceeds along the lines of the proof of Lemma B.3.2 (and Lemma B.3.3). In the REC case (and the Identifier case) we appeal to Lemma B.4.1, instead of Lemma B.3.1(1). □

The following theorem states the correctness of the encoding.

**Theorem B.4.3 (Correctness of \([\cdot] : \text{rec}_1 \rightarrow \text{rep}\).** Let \([\cdot] : \text{rec}_a \rightarrow \text{rep}\) be the encoding in Definition 8.6.3. The processes \(P\) and \([P]\) are input-output equivalent.

Proof. It follows from Lemma B.4.2 and Theorem 8.7.13. □

### B.5 Correctness of the encoding \(\text{rec}_p \rightarrow \text{rec}_d\)

The following simple observation, which states how to replace substitutions by using existential quantification and equality, is crucial for the proof of correctness of this encoding.

**Proposition B.5.1.** For any constraint \(c \in C\), \(c[y/x] = \exists x (c \wedge y = x)\).

Intuitively, the lemma below is the adaptation of the above proposition to tcc languages.

**Lemma B.5.2.** Let \([\cdot] : \text{rec}_p \rightarrow \text{rec}_d\) be the encoding in Definition 8.6.6. For all \(c \in C\) and \(P \in \text{rec}_p\):
\[
(P[y/x], c) \sim_w (\langle \text{local} \bar{x} \rangle (\langle [P] \parallel E_{y/x} \rangle), c)
\]

Proof. For the simplicity of the presentation, we assume that all the recursive definitions in \(\text{rec}_p\) have arity one and the same formal parameter, namely \(x\). It is not difficult to see that \(A(z) \stackrel{\text{def}}{=} P\) can be rewritten as \(A(x) \stackrel{\text{def}}{=} P[x/z]\), so the second assumption is not restrictive. The first assumption simplifies the presentation of the simulations below.

We observe that
\[
(\text{local} x) (\langle [P] \parallel E_{y/x} \rangle) \sim_w (\text{local} x, y = x) (\langle [P] \parallel \text{next} E_{y/x} \rangle) = \gamma'.
\]
Hence, to prove the result it suffices to give two simulations (namely, \(B \sim_w\) and \(\sim_w B^{-1}\) below) containing \((P[y/x], c), \gamma'\) and \((\gamma', P[y/x], c)\), respectively.
Let $B$ be the smallest relation s.t.,

$$((P[y/x], c), (\text{local } x, e' \land y = x) ([P] \parallel \text{next } E_{y/x}), c)) \in B,$$

whenever $P$ is a $\text{rec}_p$ process and $c = e \land e'[y/x]$.

We wish to prove that the relations $B \sim_w B^{-1}$ are simulations by using Propositions 8.7.11 and 8.7.12, respectively. Items 1 and 3 required in conditions of these propositions can be easily verified. Let us focus our attention on Item 2.

**Item 2 for $B \sim_w$.** Let $(\gamma_1, \gamma'_1)$ be a pair of configurations

$$((P[y/x], c), (\text{local } x, e' \land y = x) ([P] \parallel \text{next } E_{y/x}), c)) \in B.$$

We shall show that if $\gamma_1 \rightarrow \gamma_2$ then there must be a $\gamma'_2$ s.t, $\gamma'_1 \rightarrow^* \gamma'_2$ and $\gamma_2 B \sim_w \gamma'_2$.

Suppose that $\gamma_1 \rightarrow \gamma_2$. We shall exhibit a $\gamma'_2$ s.t $\gamma'_1 \rightarrow^* \gamma'_2$, $\gamma_2 B \sim_w \gamma'_2$. We proceed by induction (on the depth) of the inference of $\gamma_1 \rightarrow \gamma_2$.

Here we illustrate the TELL and REC cases.

**Using TELL.** We have $\gamma_1 = \langle \text{tell}(d[y/x]), c \rangle$, $\gamma_2 = \langle \text{skip}, c \land d[y/x] \rangle$, and $\gamma'_1 = \langle \text{local } x, e' \land y = x \rangle (\text{tell}(d) \parallel \text{next } E_{y/x}), c \rangle$ with $c$ being the constraint $e \land e'[y/x]$.

Using LOC together with TELL we obtain

$$\gamma'_1 \rightarrow \gamma'_2 = \langle \text{local } x, e'' \rangle (\text{skip} \parallel \text{next } E_{y/x}), c \land \exists x e'' \rangle.$$

where $e''$ is the constraint $(e' \land y = x \land d)$. But from Proposition B.5.1 $c \land \exists x (e' \land y = x \land d) = c \land d[y/x] \land e'[y/x] = c \land d[y/x]$ and thus we are done, since $[\text{skip}] = \text{skip}$.

**Using REC.** Let us confine our attention to the key case where the formal parameter of the recursive call is to be replaced with $y$.

We must therefore have $\gamma_1 = \langle A(x)[y/x], c \rangle$, $\gamma_2 = \langle Q[y/x], c \rangle$, and $\gamma'_1 = \langle \text{local } x, e' \land y = x \rangle (((\text{local } x) (A) \parallel E_{y/x}) \parallel \text{next } E_{y/x}), c \rangle$ with $c = e \land e'[y/x]$, $A(x) \overset{\text{def}}{=} Q \in D_{\text{rec}_p}$ and $A \overset{\text{def}}{=} [Q] \in D_{\text{rec}_q}$.

Using LOC together with REC, we obtain

$$\gamma'_1 \rightarrow \gamma'_2 = \langle \text{local } x, e'' \rangle (((\text{local } x) ([Q] \parallel E_{y/x})) \parallel \text{next } E_{y/x}), c \rangle$$

where $e'' = (e' \land y = x)$.

We then verify that

$$\gamma'_2 \sim_w \langle \text{local } x, e'[y/x] \land y = x \rangle ([Q] \parallel \text{next } E_{y/x}), c \rangle.$$

Since $e \land (e'[y/x][y/x]) = e \land e'[y/x] = c$, we are done.

We now turn to the Condition 2 for $\sim_w B^{-1}$. 
Item 2 for $\sim_w B^{-1}$. Let $(\gamma_1', \gamma_1)$ be a pair

$$(\langle (\text{local } x, e' \land y = x) ([P] \parallel \text{next } E_{y/x}), c \rangle, \langle P[y/x], c \rangle) \in B^{-1}.$$  

We need to show that if $\gamma_1' \rightarrow^* \gamma_1''$ then we can find a $\gamma_2$ s.t. $\gamma_1 \rightarrow^* \gamma_2$ and $\gamma_1'' \sim_w B^{-1} \gamma_2$.

Suppose that $\gamma_1' \rightarrow^n \gamma_1''$. We shall exhibit a $\gamma_2$ s.t., $\gamma_1 \rightarrow^* \gamma_2$ and $\gamma_1'' \sim_w B^{-1} \gamma_2$. The proof proceeds by induction on $n$ and on the size of the process in $\gamma_1'$. We confine ourselves to the identifier case.

Identifiers As before, we confine our attention to the case where the formal parameter of the recursive call is to be replaced with $y$.

In this case we must have $\gamma_1 = \langle A(x)[y/x], c \rangle = \langle A(y), c \rangle$ and $\gamma_1' = \langle (\text{local } x, e' \land y = x) ([A(y)] \parallel \text{next } E_{y/x}), c \rangle$. We notice that $\gamma_1' = \langle (\text{local } x, e' \land y = x) (([\langle \text{local } x \rangle (A \parallel E_{y/x})] \parallel \text{next } E_{y/x}), c) \rangle$

where $c = e \land (e'[y/x])$. $A(x) \overset{\text{def}}{=} Q \in D_{\text{rec}_p}$ and $A \overset{\text{def}}{=} [Q] \in D_{\text{rec}_i}$.

Let $\gamma_1'' = \langle (\text{local } x, e'[y/x] \land y = x) (A \parallel \text{next } E_{y/x}), c \rangle$. We verify that $\gamma_1'' \sim_w \gamma_1'$ and furthermore that $\gamma_1'' \rightarrow^m \sim_w \gamma_2'$ with $m \leq n$.

Notice that the size of the process in $\gamma_1''$ is less than the size of the process in $\gamma_1'$. Also, $e \land (e'[y/x][y/x]) = e \land e'[y/x] = c$. Hence, $\gamma_1'' B^{-1} \gamma_1$.

We can therefore use the induction to obtain $\gamma_1 = \langle A(y), c \rangle \rightarrow^* \gamma_2$ and $\gamma_2' \sim_w B^{-1} \gamma_2$, as wanted.

\[\square\]

We are now ready to state the correctness of the encoding.

**Theorem B.5.3 (Correctness of $[\cdot] : \text{rec}_p \rightarrow \text{rec}_a$).** Let $[\cdot] : \text{rec}_p \rightarrow \text{rec}_a$ be the encoding in Definition 8.6.6. The processes $P$ and $[P]$ are input-output equivalent.

**Proof.** Let us consider two arbitrary configurations $\gamma = \langle Q[\bar{y}/\bar{x}], \text{true} \rangle$ and $\gamma' = \langle (\text{local } \bar{x}, \bar{y} = \bar{x}) ([Q] \parallel \text{next } E_{\bar{y}/\bar{x}}), \text{true} \rangle$.

We observe that $\gamma \sim_w \langle A(\bar{y}), \text{true} \rangle$ and $\gamma' \sim_w \langle [A(\bar{y})], \text{true} \rangle$, for $A(\bar{y}) \overset{\text{def}}{=} Q$. Therefore, from Lemma B.5.2 and Theorem 8.7.13 we conclude that $A(\bar{y}) \sim_{\text{io}} [A(\bar{y})]$ for every definition $A(\bar{y}) \overset{\text{def}}{=} Q$ in $D_{\text{rec}_p}$.

Therefore, the theorem holds for the case in which $P$ takes the form of an identifier. As for all the other cases of $P$, we can assume, without loss of generality, that there is a definition $D_P(x) \overset{\text{def}}{=} P$ in $D_{\text{rec}_p}$. So, the results follows from the identifier case.

\[\square\]

**B.6 Correctness of $[\cdot] : \text{rep} \rightarrow \text{rec}_s$**

Here we do not use the simulation technique. Basically, the correctness follows from the fact that recursive definitions used in the encoding (Definition 8.6.7) denote recursive equations describing the input-output automata representation of $\text{rep}$ processes (see Algorithm 7.2 in Section 7.3.5).
Remark B.1. It is convenient to recall and introduce some conventions and notations. Let $\Omega = \{P\}$ where $P$ is in $\rep$. Let $M_P = A_P^\sigma$ be the automaton representing the input-output behavior of $P$ on the inputs of relevance for $P$, $\mathcal{C}(\Omega)$ (Definition 7.2.11). Let $T_P$ be the set of transitions of $M_P$. The start state of $M_P$ is $P$ and for each $e \in \mathcal{C}(\Omega)$, the transition $\langle Q, (e, e \land d), R \rangle \in T_P$ represents an observable transition $Q \xrightarrow{(e,c\land d)} R$. The set of states in $M_P$ is quotiented by the compact structural congruence $\equiv_c \mathcal{C} \sim \mathcal{I}_0$ (see Definition 7.3.7). In the rest of this section we shall consider the $\rep$ processes modulo $\equiv_c$.

Also recall that $[P] = A_P$ (Definition 8.6.7) for

$$A_P = \prod_{\langle P, (e, e \land d), R \rangle \in T_P} \text{when } e \text{ do (tell}(e \land d) \parallel O(\sqcup e, R))$$

with $\sqcup e = \bigvee_{e'' \in \{e' : e' \neq e, e' \vdash e, \langle Q, (e', e' \land d), R' \rangle \in T_P\}} e''$ where $O(\sqcup e, R) = \text{unless } \sqcup e \text{ next } A_R$ if $e \neq \text{false}$ else $O(\sqcup e, R) = \text{next } A_R$.

**Notation B.6.1.** Given $A_P$ above, let use the notation $W(e', e' \land d', R')$ as an abbreviation of the component

$$\text{when } e' \text{ do (tell}(e' \land d') \parallel O(\sqcup e', R'))$$

with $\langle P, e', e' \land d', R' \rangle \in T_P$, in the product of $A_P$.

**Remark B.2.** Let $\Omega = \{P\}$. Observe that from the automata construction and the definition of $A_P$, each $W(e', e' \land d', R')$ corresponds to a transition $P \xrightarrow{(e', e' \land d')} R'$ in $\rep$. Also since $\rep$ is deterministic, then given $e'$ (in $\mathcal{C}(\Omega)$) there must be a unique component $W(e', d', R')$ in $A_P$.

We use the following lemma relating automata and process transitions.

**Lemma B.6.2.**

$$A_P \xrightarrow{(e,c \land d)} A_R \text{ iff } \langle P, (e, e \land d), R \rangle \in T_P$$

where $e$ is such that (a) $c \vdash e$ and (b) $e \vdash e'$ for every $\langle P, (e', e' \land d'), R' \rangle \in T_P$ s.t. $c \vdash e'$.

**Proof.**

The “only if” direction. Suppose $A_P \xrightarrow{(e,c \land d)} A_R$. From this assumption and the definition of $A_P$, we can check that (a') $\langle P, (e, e \land d), R \rangle \in T_P$ with $c \vdash e$, and (b') for all $e''$ s.t., $\langle P, (e'', e'' \land d''), R'' \rangle \in T_P$, $e'' \vdash e$ and $e'' \neq e$, we have $c \not\vdash e''$. So (a') implies (a). To show (b) assume (by means of contradiction) that there exists $e'$ distinct from $e$, s.t., $\langle P, (e', e' \land d'), R' \rangle \in T_P$ and $c \vdash e'$ but $e \not\vdash e'$. Since $\mathcal{C}(\Omega)$ is closed under conjunction we have $e'' = e \land e' \in \mathcal{C}(\Omega)$. Clearly, $c \vdash e'' \vdash e$, and as we consider as inputs all the elements in $\mathcal{C}(\Omega)$ in the automata construction, there must be a $\langle P, (e'', e'' \land d''), R'' \rangle \in T_P$. This contradicts (b') above.

The “if direction”. Assume (a),(b) and that $A_P \xrightarrow{(e,c \land d)} Q$. We shall prove that $Q = A_R$ where $\langle P, e, e \land d, R \rangle \in T_P$. 
1. Notice that in the automata construction we take as inputs all constraints in \( C(\Omega) \), therefore \( e \) in the lemma is the strongest consequence of \( c \) in \( C(\Omega) \), i.e., \( e = e(\Omega) \in C(\Omega) \) (see Definition 7.2.12).

2. From (a) \( c \models e \), so \( W(e, e \land d, R) \) (Notation B.6.1) eventually tells its \( (e \land d) \). From Remark B.2 we must then have \( P \xrightarrow{e,d} R \), and hence, using Theorem 3.4.3(3) we conclude that \( P \xrightarrow{(e,d,e \land d)} R \). Hence there must be a \( W(e \land d, e \land d, R) \) in the product of \( A_P \). As the store eventually becomes \( (c \land d) \), \( W(e \land d, e \land d, R) \) will also tell \( (e \land d) \).

3. Let \( S = \{ W(e', e' \land d', R') \mid e \land d \models e' \} - \{ W(e, e \land d), W(e \land d, e \land d) \} \). As the store becomes \( (c \land d) \), every component in \( S \) will also tell its \( (e' \land d') \).

4. From the automata construction (Remark B.1) and Notation B.6.1, it follows the all the constraints \( e' \) and \( (e' \land d') \) in \( S \) are drawn from \( C(\Omega) \). Furthermore, from Remark B.2, we have \( P \xrightarrow{(e',e \land d')} R' \) for each \( W(e', e' \land d', R') \) in \( S \), and, as concluded in (2) above, \( P \xrightarrow{(e,e \land d)} R \) and \( P \xrightarrow{(e,d \land e \land d)} R \). So, since \( e \) is the strongest consequence of \( c \) in \( C(\Omega) \) (as remarked in (1)), with the help of Property 3.4.2 and the fact that \( \text{rep} \) is deterministic, we conclude that \( (e \land d) \) entails every \( (e' \land d') \) told by the components in \( S \). Hence, after the execution of all the components in \( S \), \( W(e, e \land d) \) and \( W(e \land d, e \land d) \), the store will be \( (c \land d) \).

5. From the assumptions (a) and (b) about \( e \) and the definition of \( S \), it follows that for any other component \( W(e'', e'' \land d'', R'') \) not in the set \( S \cup \{ W(e, e \land d, R), W(e \land d, e \land d, R) \} \), we have \( (c \land d) \not\models e'' \). Consequently, the final store will be \( (c \land d) \) and the only components in \( A_P \) that could trigger future activity (i.e., their “next” or “unless” guarded processes) are \( W(e, e \land d, R), W(e \land d, e \land d, R) \) and those in \( S \).

6. Notice that \( O(\lor e', R') \) takes the form \textbf{unless} \( \lor e' \textbf{ next} A_{R'} \) in every component in \( S \). (If \( e' \) were \textbf{false} then \( e \land d = \textbf{false} \) which would imply that \( W(e', e' \land d', R') \) and \( W(e \land d, e \land d, R) \) denote the same component. Thus, \( W(e', e' \land d', R') \) would be in \( S \), a contradiction.) Furthermore, we must have \( (c \land d) \models (e \land d) \models \lor e' \) for every \( e' \) in the components in \( S \) because we have \( (P, e \land d, e \land d, R) \in T_P \). Hence, none of the components in \( S \) will trigger future activity. So, we are left with \( W(e, e \land d, R) \) and \( W(e \land d, e \land d, R) \).

6.1 Suppose that \( c = \textbf{false} \). Since \( e = e(\Omega) \) we must have \( e = \textbf{false} \) (by definition \( \textbf{false} \in C(\Omega) \)), and thus \( e \land d = \textbf{false} \). Therefore, \( W(e, e \land d, R) \) and \( W(e \land d, e \land d, R) \) represent the exactly same component (Remark B.2). Since \( e = \textbf{false}, O(\lor e, R) = \textbf{next} A_R \). Therefore, \( A_R \) is the only process executed in the next time unit, i.e. \( Q = A_R \).

6.2 Now suppose that \( e \neq \textbf{false} \) and thus that \( c \neq \textbf{false} \). We have \( O(\lor e, R) = \textbf{unless} \lor e \textbf{ next} A_R \). Since \( (P, e \land d, e \land d, R) \in T_P \), we conclude that \( c \land d \models e \land d \models \lor e \). Hence, \( W(e \land d, R) \) cannot trigger its \( A_R \). So, we are left with \( W(e \land d, e \land d, R) \) only. Either (I) \( e \land d = \textbf{false} \) or (II) \( e \land d \neq \textbf{false} \). If (I) then \( O(\lor e \land d, R) = \textbf{next} A_R \), and thus \( Q = A_R \). If (II) then \( O(\lor e \land d, R) = \textbf{unless} \lor e \land d \textbf{ next} A_R \). Furthermore, \( (c \land d) \not\models \lor (e \land d) \) from our assumption about \( e \). Hence, \( Q = A_R \).
Theorem B.6.3 (Correctness of $[\cdot] : \text{rep} \to \text{rec}_s$). Let $[\cdot] : \text{rep} \to \text{rec}_s$ be the encoding in Definition 8.6.7. The processes $P$ and $[P]$ are input-output equivalent.

Proof. Let $P$ be an arbitrary process in $\text{rep}$ and let $\Omega = \{P\}$. So, we want to prove that $P \sim_\omega [P] = A_P$. It suffices to show that:

$$A_P \xrightarrow{(c,c\land d)} A_R \text{ iff } P \xrightarrow{(c,c\land d)} R \quad (B.8)$$

Consider the “only if” direction. Suppose that $A_P \xrightarrow{(c,c\land d)} A_R$. Since we consider as inputs all the elements in $C(\Omega)$, $e$ in the Lemma B.6.2 is the strongest consequence of $c$ in $C(\Omega)$, i.e., $e = c(\Omega) \in C(\Omega)$ (see Definition 7.2.12). From the automata construction, it follows that $P \xrightarrow{(c(\Omega),c(\Omega)\land d)} R$.

By appeal to Lemma 7.2.13 we conclude $P \xrightarrow{(c,d)} R$.

Consider the “if” direction. Suppose that $P \xrightarrow{(c,d)} R$. By appeal to Lemma 7.2.13 we conclude $P \xrightarrow{(c(\Omega),c(\Omega)\land d)} R$. From the automata construction, $\langle e, (c,e \land d), R \rangle \in T_P$ with $e = c(\Omega) \in C(\Omega)$. Hence, $A_P \xrightarrow{(c,c\land d)} A_R$ by using Lemma B.6.2.

$\square$
Bibliography


