Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Model Checking for the Concurrent Constraint Paradigm

Candidate:

Alicia Villanueva García

Supervisor:

María Alpuente Frasnedo
Moreno Falaschi

Author's e-mail: alicia@dimi.uniud.it

Author's address:

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

to my father
to my mother and my sister

Acknowledgments

First of all I would like to thank my advisors Moreno Falaschi and María Alpuente for their dedication. I am grateful to María who showed me the beauty of Declarative Programming and of scientific research. I am also very grateful to Moreno for his help and support during my stages in Udine.

I want to thank the people I worked with. Moreno Falaschi and Alberto Policriti were my first co-authors and guided me through the first steps of a long way that now arrives to this first milestone of my life. Gianluca Amato helped me to better understand many things by looking at them from different viewpoints.

Thanks to my colleagues in Valencia and Udine. Cèsar Ferri and Santiago Escobar were a reference point since the first instant. I will never forget the months we spent in our "black box" during the first year. I am very grateful to Carla Piazza. Carla has taught me so many things that I never will be grateful enough to her. She was my first Italian friend.

I kindly thank my friends Marco Comini and Luca di Gaspero who helped me with the formatting of this thesis. Marco helped me to acquire self-confidence and encouraged me during this last year. He has become a very special friend. Luca, as Carla, was my reference point in Italy. They always were the example to follow. I also thank Germán for his friendship. It was great to know him.

I would like to thank all the people who were in the research group of Valencia during these last years. They have always created a friendly environment inside the group.

I would also thank my friends in Valencia Carlos, Bea, Ati, Félix, Linu, Rebeca, David, Guillem and "los rapi" in general whom I will never forget. I am also grateful to Antonio. My deep deep gratitude is devoted to Raúl, who in the last months has been my principal "power source". Maybe all them do not know how much they helped me during different periods of time.

Finally, but not less important, a special thank to my family. Thanks to my mother and my sister for their unconditioned support along all these last years. They have always showed their trust in me. Perhaps they do not know how much they helped me. I must also thank my father since he taught me that there were not only games behind a computer.

April 2003,

Alicia Villanueva García

Abstract

Formal verification of temporal properties is necessary in many real applications. We can find in the literature many case studies which show us how formal verification techniques allowed scientists to find errors in systems that were thought to be correct.

Model checking is an automatic formal verification technique that, given a model of a system and a temporal formula determines if the model satisfies the formula. The main drawback of model checking is the state explosion problem.

In this thesis we consider the Concurrent Constraint (cc) paradigm to specify reactive and hybrid systems. Then, we provide the necessary formalism to verify such kind of systems. In particular, we handle three of the extended languages defined from the cc paradigm: the tcc, tccp and hcc languages. The first two languages have a discrete notion of time allowing the programmer to model reactive systems. The last one introduces a continuous notion of time which allows us to model hybrid systems.

It is well known that an appropriate denotational semantics allows us to perform very interesting analysis over languages in a simple way. In that sense, in this thesis we show that although both denotational and operational semantics were given when tcc was defined, they do not always coincide. We define a fully abstract denotational semantics (w.r.t. the operational semantics) for the tcc model. We also describe an application of our new semantics to the analysis of the expressivity power of the new construct introduced in tcc to model the timeout or preemption behaviors. We show that the new construct makes the tcc language a more powerful language than the cc model.

The main result of this thesis is the definition of a model checking algorithm for tccp programs. The idea is to exploit the good features of the cc paradigm to solve the state explosion problem of model checking. We take advantage of the *constraint* notion in order to redefine the three phases of the model checking technique. First of all we use constraints to define what a state is in the model of the system. A state of the model can be seen as a conjunction of constraints. This means that a state of our model represents a set of states of a classical *Kripke Structure*.

Furthermore, constraints are directly used in the logic that we consider, thus it is not necessary to transform our model into a Kripke Structure. For classical temporal logics, this transformation would be necessary since they could not handle our model directly. Note that we have only partial information while classical logics need full information about values of variables.

We also define a method to verify hcc programs. We show that the hcc language allows the programmer to specify hybrid systems in general and linear hybrid systems in particular. The key idea in this case is also to take advantage of the nature of

the cc paradigm. The approach presented in this thesis is the first attempt to apply the model checking technique to the hcc language.

Resumen

En muchas aplicaciones reales la verificación formal de propiedades temporales es imprescindible. En la bibliografía podemos encontrar muchos casos de estudio en los que se muestra cómo las técnicas de verificación formal han permitido detectar errores en sistemas que parecían correctos.

El *model checking* es una técnica automática de verificación formal que, dado un modelo del sistema y una fórmula temporal, comprueba si el modelo satisface la fórmula. El problema más importante del *model checking* es la explosión de estados.

El punto de partida de esta tesis es el paradigma *Concurrent Constraint* (cc) con el que es posible modelar tanto sistemas reactivos como sistemas híbridos. A partir de este paradigma, hemos definido las herramientas necesarias para la verificación formal para estos sistemas. En particular, hemos trabajado con tres extensiones temporales del modelo cc: el lenguaje tcc, tccp y el lenguaje hcc. Los dos primeros lenguajes pueden modelar sistemas reactivos, mientras que el tercer lenguaje es capaz de modelar sistemas híbridos gracias a que está definido sobre una noción de tiempo continuo.

Sabemos que una adecuada semántica denotacional permite realizar análisis muy interesantes de lenguajes y programas. Originalmente se dieron las semánticas operacional y denotacional, sin embargo estas dos semánticas no siempre son equivalentes. En esta tesis se ha definido una semántica denotacional *fully abstract* (con respecto a la semántica operacional) para el modelo de programación tcc. Además mostramos un ejemplo de cómo se puede usar esta nueva semántica denotacional para analizar la expresividad del operador de tcc que modela los conceptos de *timeout* y *preemption*. En particular, se demuestra que la introducción de dicho constructor hace al lenguaje tcc más potente que el paradigma cc.

El resultado más importante de esta tesis es la definición de un algoritmo de *model checking* para el lenguaje tccp. La idea principal consiste en explotar las características del paradigma cc para resolver (en parte) el problema de la explosión de estados. Se ha usado la noción de *constraint* (restricción) para redefinir las tres fases principales de las que consta el *model checking*. Por un lado se han definido los estados del modelo en base al concepto de restricción. De este modo, un estado de nuestro modelo representa un conjunto de estados de una estructura de Kripke tradicional. Por otro lado, la lógica que se usa para especificar la propiedad está definida con respecto al concepto de restricción directamente permitiéndonos poder manejar directamente nuestro modelo sin necesidad de transformarlo en un modelo tradicional.

En la parte final de la tesis, se ha definido un método para verificar programas especificados en el lenguaje hcc. Con hcc se pueden modelar sistemas híbridos en

general e híbridos lineales en particular. Se ha introducido un método que construye un autómata híbrido lineal a partir de la especificación hcc, el cual puede ser dado como entrada a uno de los más famosos *model checkers* del sector.

Sommario

La verifica formale di proprietà temporali è diventata essenziale per tante applicazioni reali. Si possono trovare molti esempi che ci mostrano come le tecniche di verifica formale sono riuscite a trovare errori in sistemi che si pensavano essere corretti.

Model checking è una tecnica automatica di verifica che, dato il modello del sistema e una formula temporale, verifica se il modello soddisfa la formula. Il problema più importante di questa tecnica è la esplosione del numero di stati.

Il punto di partenza di questa tesi è il paradigma *Concurrent Constraint* (cc) con il quale è possibile specificare sia sistemi reattivi che sistemi ibridi. Partendo questa base, abbiamo definito dei formalismi necessari per la verifica formale dei sistemi. Si sono considerati tre delle estensioni temporali del modello cc, ovvero i linguaggi tcc, tccp e hcc. I primi due linguaggi riescono a modellare sistemi reattivi poiché sono definiti sopra una nozione discreta di tempo, mentre il terzo linguaggio viene usato per specificare sistemi ibridi poiché usa una nozione di tempo continuo.

È ben noto che una semantica denotazionale ben definita è la base per fare delle analisi molto interessanti sui linguaggi di programmazione. Il linguaggio tcc fu definito insieme alla sua semantica operazionale e denotazionale, però queste due semantiche non coincidono sempre. In questa tesi si è definita una semantica denotazionale che è *fully abstract* rispetto alla semantica operazionale per il modello di programmazione tcc. In più si mostra un esempio di utilizzo di questa nuova semantica per analizzare il potere espressivo del costruttore di tcc che modella le nozioni di *timeout* e *preemption*. In particolare, si è dimostrato che questo operatore fa diventare il linguaggio tcc più espressivo del paradigma cc.

Il risultato più importante di questa tesi è la definizione di un algoritmo di *model checking* per i programmi specificati in tccp. Si sono utilizzate le caratteristiche del paradigma cc per dare una soluzione (parziale) al problema della esplosione degli stati. In pratica si è usata la nozione di *constraint* (vincolo) per ridefinire le tre fasi principali del *model checking*. Per prima cosa, si è definito uno stato del modello partendo del concetto di vincolo. Intuitivamente uno stato del nostro modello rappresenta un insieme di stati di una struttura di Kripke tradizionale. D'altra parte, la logica che si usa per specificare la proprietà che si vuole verificare è definita rispetto alla nozione di vincolo. Questo ci permette di non dovere trasformare il nostro modello in un modello tradizionale.

Come terzo risultato principale si è definito un metodo per la verifica di programmi specificati nel linguaggio hcc. hcc permette di modellare sistemi ibridi in generale e ibridi lineari in particolare. Si è introdotto un metodo che costruisce un'automata ibrido lineare a partire della specifica in hcc, il quale può essere dato come input a uno del più famosi *model checkers* della comunità.

Resum

En moltes aplicacions reals, la verificació formal de les propietats temporals és imprescindible. A la biografia podem trobar molts casos d'estudi en els quals es mostren com les tècniques de verificació formal han permés detectar errors als sistemes que semblaven correctes.

El *model checking* és una tècnica de verificació automàtica que, donat un model del sistema i una fórmula temporal, comproba si el model ha satistet la formula. El problema principal del model checking és l'explosió d'estats.

En aquesta tesi es fa servir el paradigma *Concurrent Constraint* (cc) per especificar sistemes reactius i sistemes híbrids. Es consideren tres dels llenguatges temporals que han estat definits a partir del paradigma cc: el llenguatge tcc, el llenguatge tccp i el llenguatge hcc. Els dos primers poden modelar sistemes reactius gràcies a que han estat definits sobre una noció de temps discret, mentre que el tercer llenguatge pot modelar sistemes híbrids, ja que fa servir una noció de temps continu.

Sabem que una adequada semàntica denotacional permet realitzar anàlisis molt interessants de llenguatges. Quan va ser introduït el llenguatge tcc es va definir la seva semàntica operacional i la seva semàntica denotacional, però aquestes dues semàntiques a voltes no coincideixen. En aquesta tesi s'ha definit una semàntica denotacional *fully abstract* respecte de la semàntica operacional per el llenguatge tcc. A més mostrem un exemple de com es pot fer servir aquesta nova semàntica denotacional per analitzar el poder expressiu del constructor de tcc que modela els conceptes de *timeout* i *preemption*. S'ha demostrat que la introducció d'aquest constructor fa al llenguatge tcc més potent que el paradigma cc.

El resultat més important d'aquesta tesi és la definició d'un algorisme de *model checking* per al llenguatge tccp. La idea és explotar les característiques del paradigma cc per a resoldre (en part) el problema de l'explosió d'estats. S'ha emprat la noció de *constraint* per a redefinir dues de les tres fases principals del *model checking*. Per una banda s'ha definit un estat del model sobre la base del concepte de constraint. D'un mode intuïtiu, un estat del nostre model representa un conjunt d'estats d'una estructura de Kripke tradicional. Per l'altra banda, la lògica que s'utilitza per especificar la propietat està definida respecte al concepte de *constraint* directament. Això ens permet no haver de transformar el nostre model en un model tradicional.

Per últim, es defineix un mètode per a verificar programes especificats en el llenguatge hcc. Amb hcc es poden modelar sistemes híbrids en general, i híbrids linials en particular. En aquesta tesi s'introdueix un mètode que construeix un autòmata híbrid linial a partir de l'especificació, el qual pot ser donat com a entrada

a un dels més famosos model checkers del sector.

Contents

List of Figures

Introduction

Computer Science can help in the design and realization of the algorithmic solution to an application problem. This consideration has led to the field of specification and verification of programs. The idea is to have mechanisms to check that systems satisfy all requirements of the user. Thus, such mechanisms must be able to translate specifications in high-level programming languages into low-level algorithms, and also be able to check that the result of this transformation is correct.

Formal verification can prove that some properties are satisfied. In many industrial applications it is not enough to approximate the correctness of the system, but some timing and properties must absolutely be guaranteed. This importance is well remarked in the literature. In [CGP99] and [MP95] we can find some significant examples of error cases. Such practical examples show how formal verification techniques allows one to find errors in systems that were thought to be correct.

Model checking is a fully automatic technique that, given a model of a system and a temporal formula determines if the model satisfies the formula. This technique was defined for finite-state systems. However, we show in this thesis how in the last years many researchers have studied how to extend this technique to infinite-state systems.

In this thesis we tackle the problem of applying model checking to *real* languages, i.e., languages that were previously defined to specify reactive and hybrid systems. This is a first difference from the classical approaches. Usually, for each approach, the language for the specification of the system is defined ad hoc for a very specific class of systems. In our case it is possible to specify finite *and* infinite-state systems, thus if we do not want to impose a restriction on the expressiveness of the language, we have to consider the problem of model checking for infinite-state systems as well.

In particular, the three languages that we consider belong to the *concurrent constraint paradigm* (cc). Languages of this paradigm are parametric w.r.t. a notion of *constraint system* and the computational model works by asking and telling information to a shared *store*.

Timed concurrent constraint language (tccp) and *timed concurrent constraint* (tcc) paradigm extend the cc paradigm with a notion of discrete time which allows one to model reactive systems. The *hybrid* cc language (hcc) is defined over a notion of continuous (or dense) time, thus it is able to model hybrid systems.

Analysis of programming languages or programs is another important technique, which allows one to compare languages or check if a program satisfies some general property (termination, confluence, etc.). One of the most important features of the declarative paradigm is the fact that it is always possible to define appropriate semantics for the analysis of programs. Often it is easier to handle such semantics than those defined for imperative languages. Therefore, it is crucial to have semantics which represent the exact observable behavior of programs

I.1 A Model for Concurrency

Concurrent systems are systems consisting of multiple computing *agents* (also called *processes*) that interact among each other. Some examples of these systems are communication systems based on *message-passing*, communication systems based on *shared-variables*, *synchronous* systems, *reactive* systems, *timed* systems, *mobile* systems or *secure* systems.

There are many models for concurrent systems. Some examples of synchronous models are the process calculi of Milner's CCS [Mil89], Hoare's CSP [Hoa85] or the ACP of Bergstra and Klop [BK85]. The model which we consider in this thesis is the model for concurrency that was defined in [Sar89].

The *Concurrent Constraint programming* paradigm (cc) was defined by Saraswat and Rinard in [Sar89, Sar93, SR90] as a simple and powerful model of concurrent computation. In such computational model the notion of *store-as-valuation* from von Neumann was replaced with the notion of *store-as-constraint*. Basically, the model evolves monotonically by accumulating information in the store.

The cc paradigm is parametric w.r.t. a *constraint system*. A constraint system can be seen as a partial information system (see [SRP91]). Thus, whereas in the von Neumann model we have always the specific value of variables, in this computational model we have partial information about the value of variables of the system.

A few years later Saraswat, Jagadeesan and Gupta defined an extension over time of the cc paradigm. This new language, called *temporal concurrent constraint* (tcc) language ([SJG94a, SJG94b]) was inspired by synchronous languages such as ESTEREL [Ber00], Lustre [HCRP91] or SIGNAL [GGBM91]. The key idea was to introduce a notion of *discrete* time and some constructs which allow us to model notions such as *timeout* or *weak preemption*.

Therefore, tcc is able to specify *reactive systems* that can be defined as those systems which evolve along the time by interacting with their environment. In particular tcc is suitable to model embedded systems (a subclass of reactive systems).

Other extensions over time were presented to improve the expressiveness of the model. For example, the *Timed Default Concurrent Constraint Programming* was defined in [SJG96]. This language allows one to model also *strong preemptions*. Moreover, in 1998 Gupta, Jagadeesan and Saraswat presented a language which incorporates a notion of *continuous* (or *dense*) time to the cc model: the *hybrid* cc (hcc) language.

The hcc language is able to model *hybrid systems* which can be defined as those systems that have a continuous behavior controlled by a discrete component. For example, a thermostat can be seen as an hybrid system. It has a continuous variable modelling the temperature, and a discrete control that turn-on or turn-off the system depending on the limits established for the temperature value. [GSS95] shows some applicative examples for both the timed default concurrent constraint programming language and the hcc language.

In [BGM00], Frank de Boer, Gabbrielli and Meo presented a different approach to extend the cc paradigm with a notion of discrete time. *Temporal concurrent constraint* language (tccp) was inspired by the process algebra model. As a major property, the language is non deterministic and monotone, thus it is possible to model more complex (reactive) systems.

As we have said before, reactive systems are programs that interact with their environment along the time, thus the concept of termination loses significance. Reactive systems are essentially different from the functional systems whose behavior is simply described as a transformation of the input into the output. Moreover, in reactive systems it is important to capture when events or stimuli are presented whereas in functional systems it does not matter when the input is given.

Reactive systems usually are specified as concurrent systems where the environment is modelled as a concurrent process. Therefore, concurrency is a very important notion for such systems. Moreover, due to the progress of new technologies such as internet or mobile systems, concurrency has become more and more important.

The most recent defined model in the cc family is the ntcc language [PV01]. Palamidessi and Valencia defined this language essentially as an extension of the tcc. They introduced the notion of non-determinism into the tcc model of Saraswat *et al.*.

In the following chapters we explain more in detail some of the cited cc languages. In particular we describe the tcc language of Saraswat *et al.*, the tccp language defined by de Boer *et al.* and the hcc language of Gupta *et al.*.

I.2 Formal Verification

Program verification consists in formally proving that a program satisfies some logical specification. Formal verification can be applied to hardware or software systems. In this thesis we are interested in software verification. There are many techniques which can be applied to solve the problem of software verification. However there is no general method which is able to verify any software system.

Historically, *theorem proving* was the first formal verification method studied in the literature (see [Flo67, Hoa69]). This is a deductive method which is guided by the user and must be performed essentially by hand. The success of deductive proofs depends a lot on the capability of the user, thus theorem proving must be applied by expert people.

Another method which is widely used by the research community is the *testing* technique. This technique cannot be considered a formal verification method since it is based on the analysis of only some of the executions of programs. Therefore, using this technique it is not possible to ensure the total absence of errors in the program. Testing is a very simple technique which can be used by non expert people in mathematics or logic. This is the reason why it has been widely used to improve the quality of software. In Chapter 6 we explain the features of these two techniques giving more details.

The third and most important verification method in this thesis is the *model checking* technique. This technique allows us to check if a system satisfies a specific property in an automatic way. Model checking is based on a very simple idea: to check that the execution sequences of the system are a model of the formula representing the property. Model checking was originally defined for finite-state systems since it is based in an exhaustive analysis of the state-space. However, in the last years many researches have tried to extend the technique to infinite-state systems. We introduce the background of the model checking technique in Chapter 6.

I.3 Verification and Constraints

In this thesis we are very interested in two important concepts: *verification* and *constraints*. There are several verification methods which combine these two notions. These approaches define methodologies or algorithms that allow us to apply the model checking technique to concurrent, reactive or even hybrid systems. It is also possible to verify some subclasses of infinite-state systems.

In [DP99, DP01], Delzanno and Podelski presented a method that allows us to verify a communication protocol with an *infinite* number of states. In particular, they proved that a client-server protocol is correct for an arbitrary number of processes (clients). Classical approaches could prove the same properties but only for a specific number of clients or processes.

The notion of constraint plays a very important role in the approach of Delzanno and Podelski. The idea is to translate concurrent systems into CLP programs and verify safety and liveness properties over such CLP programs.

Another interesting approach which combines verification and constraints can be found in [EM97]. Here a semidecision algorithm that uses constraint programming in order to verify 1-safe Petri nets is introduced. Actually, while in [DP01, DP99], constraints are used as an abstract representation of the set of system states, in [EM97] constraint programming is used to solve linear constraints in the implementation of the algorithm.

In [FPV00a, FPV00b], the authors presented a framework that allowed us to apply the model checking technique to tcc programs. Actually, in these works it is only presented in detail how to construct automatically a model of a timed concurrent constraint program.

We have seen that constraints can be used in any level or phase of the formal verification process. In [DP99, DP01] constraints were used to model the problem, in [FPV00a, FPV00b] the specification language is defined over an underlying constraint systems while in [EM97] constraints are used for the implementation of the model checking algorithm.

I.4 The Thesis Approach

In this thesis we intend to take advantage of some features of cc languages, i.e., languages defined within the concurrent constraint paradigm. Our final goal is to define a formal verification method for reactive and hybrid systems. We develop in detail the model checking algorithm for the tccp language. Then, we sketch a first approach to the verification of hybrid systems specified in hybrid cc. The key idea in both methods is to take advantage of the constraint nature underlying the cc paradigm.

The declarative nature of tccp and hcc, and the notion of time introduced directly into their semantics allows us to reduce the number of states of the system which is the main problem of model checking algorithms.

In particular, the introduction of a built-in notion of time makes it reasonable to restrict the verification to a *finite* interval of time, i.e., the interval of time that we want to analyze. A precondition for this approach is, clearly, that the user should know and provide the interval on which the verification should be carried out. This seems a reasonable assumption since usually is the specifier who wants to verify the correctness of his software.

tccp Model Checking

The tccp model checking problem consists in developing the model checking techniques for tccp programs. In this thesis we describe an automatic verification method to verify tccp programs based on the classical model checking algorithm for LTL. We use the notion of constraint in the different phases of the verification process.

First of all, we use constraints in the *automatic* construction of the model. Constraints can represent in a compact way a set of possible variable values (i.e., a set of states if we use the classical notion of state).

Then, in the second phase we take advantage of constraints for specifying the property by using a logic that is defined to handle constraints. Such logic is presented in [BGM01] and is able to work over constraints in a very intuitive way.

The last phase of the model checking technique consists in defining an algorithm that determines if the system satisfies the property. We use the two outputs of the previous phases in order to adapt the classical algorithm defined for LTL to our framework.

This new algorithm can be defined thanks to the fact that we use a logic that works over constraints. This logic allows us to check properties directly over the tccp Structure that we define. Since this structure is formed by states that contain constraints and can be seen as sets of classical states, it would not be possible to use a classical temporal logic directly. To the best of our knowledge this is the first model checking approach for systems specified with cc languages.

Our approach is not able to verify *all* infinite-state systems, but only a subclass of such systems. Actually we can verify a very similar subclass of infinite-state systems as [DP99, DP01]. For the other systems, we are able to verify an over-approximation of the system.

hcc Model Checking

The second contribution regarding model checking presented in this thesis is the first attempt to apply the model checking technique to the hybrid cc language. We show how some natural characteristics of the language allow us to define a method to verify linear hybrid systems.

In particular, we have defined a graph structure which represents the behavior of systems specified in hcc. Then we have transformed this graph structure into a linear hybrid automaton which can be given as input to the HyTech model checker. HyTech is one of the most popular model checkers for linear hybrid systems in the literature. Actually, although we could model any hcc program using the hcc Structure, in this thesis we have limited ourselves to linear hybrid systems since HyTech is able to handle only such subclass of hybrid systems.

Semantics for tcc

The third problema that we tackle in this thesis regards the tcc language. Although we have mentioned it at this last point if the section, it is presented in the first part of this dissertation.

In the classical literature it is well known the importance to have a denotational semantics of programming languages which is fully abstract w.r.t. the operational semantics. This is especially crucial for languages such as tcc where more than one transition relation is necessary to describe the operational semantics.

In general it is easier to analyze and to compare the characteristics of languages by using the denotational semantics. However, if the denotational semantics is not fully abstract w.r.t. the operational behavior, then we can only express over-approximations of the properties of the analyzed languages.

The tcc language was defined by Saraswat *et al.* as an extension over time of the cc paradigm. A notion of discrete time was introduced in the semantics of the language. However, in [NPV02b] was shown that the denotational semantics given in [SJG94a] corresponds to the operational behavior only for a subclass of tcc programs. In particular, we have different semantics when we try to derive *negative information* from a local variable. Negative information can be seen as the capability to execute some process when some event does not occur.

In this thesis we define a new denotational semantics which is fully abstract w.r.t. the operational semantics given in [SJG94a]. In particular we redefine the semantics for the construct which models the existential quantification over variables (i.e., it makes a variable local in a process) and we prove the correctness and completeness of the new semantics w.r.t. the operational behaviour.

As a first application of this new semantics, we also analyze the expressiveness of the construct that is able to capture the negative information of systems in tcc.

I.5 The Thesis Overview

The thesis is organized as follows:

In Chapter 1 we introduce informally the basic concepts and terminologies used in this thesis. We present in a unique chapter all the needed terminologies and notations used for both the definition of the model checking techniques and the definition of the new semantics.

In Chapter 3 the tcc language is presented in detail giving both the operational and the denotational semantics. The operational semantics is given following a different notation from the one presented in [SJG94a]. Finally, we show which kind of problems we intend to solve by introducing the new denotational semantics.

In Chapter 4 the new denotational semantics for tcc is introduced. Moreover, we show the correctness and completeness of the semantics w.r.t. the operational behavior. In the last part of this chapter we show a first application of the new denotational semantics in order to analyze the expressive power of the agent which is able to capture the negative information of the system.

In Chapter 5 we introduce the tccp language defined in [BGM00]. We discuss the main characteristics of the language and present the original operational semantics. Moreover, we show an example of an application that can be modelled using this language.

In Chapter 6 the model checking technique is described. Actually, we describe the classical approaches and we discuss the different research lines that are presented in the literature. Moreover, in the last part of the chapter we describe our approach for tccp and hcc.

Chapter 8 presents the first main task of the model checking technique. It is described how we can construct a model of the system behavior (similar to the *Kripke Structure* used in the classical approaches) from the specification of the system written in tccp.

Chapter 9 describes the logic that we use in our framework. Moreover a comparison with classical logics is done and some examples showing the expressiveness of the logic are presented.

In Chapter 11 the algorithm that corresponds to the last task of the model checking technique is defined. Actually, this algorithm is a version of the classical model checking algorithm for linear time logic but adapted to our framework which presents some additional difficulties. In particular we must pay attention to agents and logic operators which existential quantify constraints. Proofs of the correctness of the algorithm are given in the last part of the chapter and an example is presented.

Chapter 12 describes the last considered language, the hcc language which is able to model hybrid systems.

In Chapter 13 a first approach to the problem of model checking for hcc programs is presented. An automatic transformation from hcc specifications to linear hybrid automata is showed.

Finally, in Chapter 14 some conclusions are drawn.

1

Preliminaries

In this chapter we introduce the notations that we use in this thesis. Essentially we present the informal, logical and set-theoretic notations and concepts that we use to write down and reason about our ideas. The chapter is simply presented by using an informal extension of our everyday language to talk about mathematical objects like sets. For the terminology not explicitly shown and algebraic notation, the reader can consult [Man74, BM65, Bir67].

1.1 Basic Set Theory

To define the basic notions that we use the standard (meta) logical notation to denote conjunction, disjunction, quantification and so on (*and*, *or*, *for each*, ...). We will use some informal logical notation in order to stop our mathematical statements getting out of hand. For statements (or assertions) A and B, we will commonly use abbreviations like:

A, B for (A and B), the conjunction of A and B,

$A \Longrightarrow B$ for (A implies B), which means (if A then B),

$A \Longleftrightarrow B$ for (A if and only if B), which expresses the logical equivalence of A and B.

We will also make statements by forming disjunctions (A or B), with the self-evident meaning, and negations (not A), sometimes written $\neg A$, which is true if and only if A is false. It is a tradition to write $x \not< y$ instead of $\neg(x < y)$.

A statement like $P(x, y)$, which involves variables x, y, is called a predicate (or property, or relation, or condition) and it only becomes true or false when the pair x, y stands for particular things. We use logical quantifiers $\exists$ (read "there exists") and $\forall$ (read "for all") to write assertions like $\exists x.\, P(x)$ as abbreviating "for some x, $P(x)$" or "there exists x such that $P(x)$", and $\forall x.\, P(x)$ as abbreviating "for all x, $P(x)$" or "for any x, $P(x)$". The statement $\exists x, y, \ldots, z.\, P(x, y, \ldots, z)$ abbreviates $\exists x.\exists y. \cdots \exists z.\, P(x, y, \ldots, z)$, whereas the statement $\forall x, y, \ldots, z.\, P(x, y, \ldots, z)$ abbreviates $\forall x.\forall y. \cdots \forall z.\, P(x, y, \ldots, z)$. In order to specify a set S over which a quantifier ranges, we write $\forall x \in S.\, P(x)$ instead of $\forall x.\, x \in S \implies P(x)$, and $\exists x \in S.\, P(x)$ instead of $\exists x.\, x \in S, P(x)$.

1.1.1 Sets

Intuitively, a set is an (unordered) collection of objects, which are elements (or members) of it. We write $a \in S$ when a is an element of the set S. Moreover, we write $\{a, b, c, \ldots\}$ for the set of elements $a, b, c, \ldots$.

A set S is said to be a subset of a set S', written $S \subseteq S'$, if and only if every element of S is an element of S', i.e., $S \subseteq S' \iff \forall z \in S.\, z \in S'$. A set S is said to be a finite subset of a set S', written $S \subseteq_f S'$, if S is a subset of S' and S has a finite number of elements. A set is determined solely by its elements in the sense that two sets are equal if and only if they have the same elements. So, sets S and S' are equal, written $S = S'$, if and only if every element of S is an element of S' and vice versa.

Sets and Properties

A set can be determined by a property P. We write $S := \{x \mid P(x)\}$, meaning that the set S has as elements precisely all those x for which $P(x)$ is true. We will not be formal about it, but we will avoid trouble like Russell's paradox and will have at the same time a world of sets rich enough to support most mathematics. This will be achieved by assuming that certain given sets exist right from the start and by using safe methods for constructing new sets.

We write $\emptyset$ for the null or empty set and $\mathbb{N}$ for the set of natural numbers $0, 1, 2, \ldots$.

The cardinality of a set S is denoted by $|S|$. A set S is called *denumerable* if $|S| = |\mathbb{N}|$ and *countable* if $|S| \leq |\mathbb{N}|$.

Constructions on Sets

Let S be a set and $P(x)$ be a property. By $\{x \in S \mid P(x)\}$ we denote the set $\{x \mid x \in S, P(x)\}$. Sometimes, we will use a further abbreviation. Suppose $E(x_1, \ldots, x_n)$ is some expression which for particular elements $x_1 \in S_1, \ldots, x_n \in S_n$ yields a particular element and $P(x_1, \ldots, x_n)$ is a property of such $x_1, \ldots, x_n$. We use $\{E(x_1, \ldots, x_n) \mid x_1 \in S_1, \ldots, x_n \in S_n, P(x_1, \ldots, x_n)\}$ to abbreviate $\{y \mid \exists x_1 \in S_1, \ldots, x_n \in S_n.\, y = E(x_1, \ldots, x_n), P(x_1, \ldots, x_n)\}$.

The *powerset* of a set S, $\{S' \mid S' \subseteq S\}$, is denoted by $\wp(S)$. The set of all finite subsets of S is denoted by $\wp_f(S)$.

Let I be a set. By $\{x_i\}_{i \in I}$ (or $\{x_i \mid i \in I\}$) we denote the set of (unique) objects x_i, for any $i \in I$. The elements x_i are said to be *indexed* by the elements $i \in I$.

The *union* of two sets is $S \cup S' := \{a \mid a \in S \text{ or } a \in S'\}$. Let $\mathcal{S}$ be a set of sets, $\bigcup \mathcal{S} = \{a \mid \exists S \in \mathcal{S}.\, a \in S\}$. When $\mathcal{S} = \{S_i\}_{i \in I}$, for some indexing set I, we write $\bigcup \mathcal{S}$

as $\bigcup_{i \in I} S_i$. The *intersection* of two sets is $S \cap S' := \{a \mid a \in S, a \in S'\}$. Let $\mathcal{S}$ be a nonempty set of sets. Then $\bigcap \mathcal{S} := \{a \mid \forall S \in \mathcal{S}.\, a \in S\}$. When $\mathcal{S} = \{S_i\}_{i \in I}$ we write $\bigcap \mathcal{S}$ as $\bigcap_{i \in I} S_i$.

The *cartesian product* of S and S' is the set $S \times S' := \{(a, b) \mid a \in S, b \in S'\}$, the set of ordered pairs of elements with the first from S and the second from S'. More generally $S_1 \times S_2 \times \cdots \times S_n$ consists of the set of n-tuples $(x_1, \ldots, x_n)$ with $x_i \in S_i$ and S^n denotes the set of n-tuples of elements in S.

$S \setminus S'$ denotes the set where all the elements from S which are also in S' have been removed, i.e., $S \setminus S' := \{x \mid x \in S, x \notin S'\}$.

1.1.2 Multisets

Intuitively, a *multiset* is a set which can contain identical elements a finite number of times [Knu80]. A multiset can be specified by listing all the occurrences of its elements. In order to distinguish multisets from sets, we will write multisets within double braces. For instance,

$$\{\{a, a, b, b, b, c\}\}$$

denotes the multiset containing two occurrences of a, three of b and one occurrence of c. The order in which the occurrences are listed is immaterial. However, the multiplicity of occurrences is relevant. The empty multiset is denoted by $\varnothing$.

If A and B are multisets, so are $A \uplus B$ and $\cap\!\!\!\!\!{\scriptstyle +}$. An element occurring n times in A and m times in B occurs exactly $n + m$ times in $A \uplus B$ and exactly $\min(n, m)$ times in $\cap\!\!\!\!\!{\scriptstyle +}$. $\uplus$ and $\cap\!\!\!\!\!{\scriptstyle +}$ are commutative and associative, moreover $\uplus$ distributes over $\cap\!\!\!\!\!{\scriptstyle +}$ and $\cap\!\!\!\!\!{\scriptstyle +}$ is idempotent. The absorption law $A \cap\!\!\!\!\!{\scriptstyle +} (A \uplus B) = A$ is satisfied; $\varnothing$ is an identity for $\uplus$, whereas it is a zero for $\cap\!\!\!\!\!{\scriptstyle +}$.

The *cardinality* of a multiset M is the number of its elements' occurrences and is denoted by $\| M \|$, thus

$$\| \{\{a, a, b, b, b, c\}\} \| = 6$$

1.1.3 Relations and Functions

A *binary relation* between S and S' ($R : S \times S'$) is an element of $\wp(S \times S')$. We write $x \mathrel{R} y$ for $(x, y) \in R$.

A *partial function* from S to S' is a relation $f : S \times S'$ for which $\forall x, y, y'.\, (x, y) \in f, (x, y') \in f \Longrightarrow y = y'$. By $f : S \rightharpoonup S'$ we denote a partial function of the set S (the *domain*) into the set S' (the *range*). The set of all partial functions from S to S' is denoted by $[S \rightharpoonup S']$. Moreover, we use the notation $f(x) = y$ when there is a y such that $(x, y) \in f$ and we say $f(x)$ is defined, otherwise $f(x)$ is undefined. Sometimes, when $f(x)$ is undefined, we write $f(x) = \aleph$, where $\aleph$ denotes the undefined element. For each set S we assume that $\aleph \subseteq S$, $\aleph \cup S = S$ and $\emptyset \not\subseteq \aleph$. This will be formally motivated in Section 1.2.1.

Given a partial function $f : S \rightharpoonup S'$, the sets $\mathrm{supp}(f) := \{x \in S \mid f(x) \text{ is defined}\}$ and $\mathrm{img}(f) := \{f(x) \in S' \mid \exists x \in S.\, f(x) \text{ is defined}\}$ are, respectively, the *support* and the *image* of f. A partial function is said to be *finite-support* if $\mathrm{supp}(f)$ is finite. Moreover, it is said to be *finite* if both $\mathrm{supp}(f)$ and $\mathrm{img}(f)$ are finite. In the following, we will often use finite-support partial functions. Hence, to simplify the notation, by

$$f := \begin{cases} v_1 \mapsto r_1 \\ \quad\vdots \\ v_n \mapsto r_n \end{cases}$$

we will denote (by cases) any function f which assumes on input values $v_1, \ldots, v_n$ output values $r_1, \ldots, r_n$ and is otherwise undefined. Furthermore, if the support of f is just the singleton $\{v\}$, we will denote it by $f := v \mapsto r$.

A (total) function f from S to S' is a partial function from S to S' such that, for all $x \in S$, there is some $y \in S'$ such that $f(x) = y$. That is equivalent as saying that f is total if $\mathrm{supp}(f) = S$. Although total functions are a special kind of partial function, it is a tradition to understand something described as simply a function to be a total function. So we will always say explicitly when a function is partial. To indicate that a function f from S to S' is total, we write $f : S \to S'$. Moreover, the set of all (total) functions from S to S' is denoted by $[S \to S']$.

A function $f : S \to S'$ is *injective* if and only if for each $x, y \in S$ if $f(x) = f(y)$ then $x = y$. f is *surjective* if and only if for each $x' \in S'$ there exists $x \in S$ such that $f(x) = x'$.

We denote by $f = g$ the extensional equality, i.e., for each $x \in S$, $f(x) = g(x)$. Furthermore, $g := f[{}^{v}\!/\!_{x}]$ denotes the function g which differs from f only for the assignment of v to x, i.e., $g(x) = v$ and, for each $y \neq x$, $g(y) = f(y)$.

Lambda Notation

It is sometimes useful to use the lambda notation to describe functions. It provides a way of referring to functions without having to name them. Suppose $f : S \to S'$ is a function which, for any element $x \in S$, gives a value $f(x)$ which is exactly described by expression E, probably involving x. Then we can write $\lambda x \in S.\, E$ for the function f. Thus, $(\lambda x \in S.\, E) := \{(x, E[x]) \mid x \in S\}$ and so $\lambda x \in S.\, E$ is just an abbreviation for the set of input-output values determined by the expression $E[x]$.

Composing Relations and Functions

We compose relations, and so partial and total functions, $R : S \times S'$ and $Q : S' \times S''$ by defining their *composition* (a relation between S and S'') by $Q \circ R := \{(x, z) \in S \times S'' \mid y \in S', (x, y) \in R, (y, z) \in Q\}$. R^n is the relation

$$\underbrace{R \circ \cdots \circ R}_{n},$$

i.e., $R^1 := R$ and (assuming R^n is defined) $R^{n+1} := R \circ R^n$. Each set S is associated with an identity function $Id_S := \{(x, x) \mid x \in S\}$, which is the neutral element of $\circ$. Thus we define $R^0 := Id_S$.

The *transitive and reflexive closure* R^* of a relation R on S is $R^* := \bigcup_{i \in \mathbb{N}} R^i$.

The *function composition* of $g : S \rightharpoonup S'$ and $f : S' \rightharpoonup S''$ is the partial function $f \circ g : S \rightharpoonup S''$, where $(f \circ g)(x) := f(g(x))$, if $g(x)$ (first) and $f(g(x))$ (then) are defined, and it is otherwise undefined. When it is clear from the context $\circ$ will be omitted.

A function $f : S \rightarrow S'$ is *bijective* if it has an *inverse* $g : S' \rightarrow S$, i.e., if and only if there exists a function g such that $g \circ f = Id_S$ and $f \circ g = Id_{S'}$. Then the sets S and S' are said to be in 1-1 correspondence. Any set in 1-1 correspondence with a subset of natural numbers $\mathbb{N}$ is said to be countable. Note that a function f is bijective if and only if it is injective and surjective. A function is called a *homomorphism* on an algebra Q in symbols $h \in Q$ if $dom(h) = A$ and for any $x, x', y, y' \in A$, the formulas $h(x) = h(x')$ and $h(y) = h(y')$ imply $h(x + y) = h(x' + y')$.

Direct and Inverse Image of a Relation

We extend relations, and thus partial and total functions, $R : S \times S'$ to functions on subsets by taking $R(X) := \{y \in S' \mid \exists x \in X. (x, y) \in R\}$ for $X \subseteq S$. The set $R(X)$ is called the *direct image* of X under R. We define $R^{-1}(Y) := \{x \in S \mid \exists y \in Y. (x, y) \in R\}$ for $Y \subseteq S'$. The set $R^{-1}(Y)$ is called the *inverse image* of Y under R. Thus, if $f : S \rightharpoonup S'$ is a partial function, $X \subseteq S$ and $X' \subseteq S'$, we denote by $f(X)$ the *image* of X under f, i.e., $f(X) := \{f(x) \mid x \in X\}$ and by $f^{-1}(X')$ the inverse image of X' under f, i.e., $f^{-1}(X') := \{x \mid f(x) \in X'\}$.

Equivalence Relations and Congruences

An *equivalence relation* $\approx$ on a set S is a binary relation on S ($\approx : S \times S$) such that, for each $x, y, z \in S$,

$$\begin{aligned} & x\ R\ x && \text{(reflexivity)} \\ & x\ R\ y \Longrightarrow y\ R\ x && \text{(symmetry)} \\ & x\ R\ y, y\ R\ z \Longrightarrow x\ R\ z && \text{(transitivity)} \end{aligned}$$

The *equivalence class* of an element $x \in S$, with respect to $\approx$, is the subset $[x]_\approx := \{y \mid x \approx y\}$. When clear from the context we abbreviate $[x]_\approx$ by $[x]$ and often abuse notation by letting the elements of a set denote their correspondent equivalence classes. The *quotient set* $S/_\approx$ of S modulo $\approx$ is the set of equivalence classes of elements in S (w.r.t. $\approx$).

An equivalence relation $\approx$ on S is a *congruence* w.r.t. a partial function $f : S^n \rightharpoonup S$ if and only if, for each pair of elements a_i, $b_i \in S$ such that $a_i \approx b_i$, (if $f(a_1, \ldots, a_n)$ is defined then also $f(b_1, \ldots, b_n)$ is defined and)

$$f(a_1, \ldots, a_n) \approx f(b_1, \ldots, b_n).$$

Then, we can define the partial function $f_{\approx} : (S/_{\approx})^n \rightharpoonup S/_{\approx}$ as

$$f_{\approx}([a_1]_{\approx}, \ldots, [a_n]_{\approx}) := [f(a_1, \ldots, a_n)]_{\approx},$$

since, given $[a_1]_{\approx}, \ldots, [a_n]_{\approx}$, the class $[f(a_1, \ldots, a_n)]_{\approx}$ is uniquely determined independently of the choice of the representatives $a_1, \ldots, a_n$.

1.2 Domain Theory

We will present here the (abstract) concepts of complete lattices, continuous functions and fixpoint theory, which are the standard tools of denotational semantics.

1.2.1 Complete Lattices and Continuous Functions

A binary relation $\leq$ on S ($\leq$: $S \times S$) is a *partial order* if, for each $x, y, z \in S$,

$$\begin{array}{ll} x \leq x & \text{(reflexivity)} \\ x \leq y, y \leq x \Longrightarrow x = y & \text{(antisymmetry)} \\ x \leq y, y \leq z \Longrightarrow x \leq z & \text{(transitivity)} \end{array}$$

A *partially ordered set* (poset) $(S, \leq)$ is a set S equipped with a partial order $\leq$. A set S is *totally ordered* if it is partially ordered and, for each $x, y \in S$, $x \leq y$ or $y \leq x$. A *chain* is a (possibly empty) totally ordered subset of S.

A *preorder* is a binary relation which is reflexive and transitive. A preorder $\leq$ on a set S induces on S an equivalence relation $\approx$ defined as follows: for each $x, y \in S$,

$$x \approx y \Longleftrightarrow x \leq y, y \leq x.$$

Moreover, $\leq$ induces on $S/_{\approx}$ the partial order $\leq_{\approx}$ such that, for each $[x]_{\approx}, [y]_{\approx} \in S/_{\approx}$,

$$[x]_{\approx} \leq_{\approx} [y]_{\approx} \Longleftrightarrow x \leq y.$$

A binary relation $<$ is *strict* if and only if it is anti-reflexive (i.e., not $x < x$) and transitive.

Given a poset $(S, \leq)$ and $X \subseteq S$, $y \in S$ is an *upper bound* for X if and only if, for each $x \in X$, $x \leq y$. Moreover, $y \in S$ is the *least upper bound* (called also join) of X, if y is an upper bound of X and, for every upper bound y' of X, $y \leq y'$. A least upper bound of X is often denoted by $\mathrm{lub}_S X$ or by $\bigsqcup_S X$. We also write $\bigsqcup_S\{d_1, \ldots, d_n\}$ as $d_1 \sqcup_S \cdots \sqcup_S d_n$. Dually an element $y \in S$ is a *lower bound* for X if and only if, for each $x \in X$, $y \leq x$. Moreover, $y \in S$ is the *greatest lower bound* (called also meet) of X, if y is a lower bound of X and for every lower bound y' of X, $y' \leq y$. A greatest lower bound of X is often denoted by $\mathrm{glb}_S X$ or by $\bigsqcap_S X$. We also write $\bigsqcap_S\{d_1, \ldots, d_n\}$ as $d_1 \sqcap_S \cdots \sqcap_S d_n$. When it is clear from the context, the subscript S will be omitted. Moreover $\bigsqcup\{D_i\}_{i \in I}$ and $\bigsqcap\{D_i\}_{i \in I}$ can be denoted by $\bigsqcup_{i \in I} D_i$ and $\bigsqcap_{i \in I} D_i$. It is easy to check that if lub and glb exist, then they are unique.

Complete Partial Orders and Lattices

A *direct set* is a poset in which any subset of two elements (and hence any finite subset) has an upper bound in the set. A *complete partial order* (*CPO*) S is a poset such that every chain D has the least upper bound (i.e., there exists $\bigsqcup D$). Notice that any set ordered by the identity relation forms a *CPO*, of course without a bottom element. Such *CPO*s are called discrete. We can add a bottom element to any poset $(S, \leq)$ which does not have one (even to a poset which already has one). The new poset $S_\perp$ is obtained by adding a new element $\perp$ to S and by extending the ordering $\leq$ as $\forall x \in S. \perp \leq x$. If S is a discrete *CPO*, then $S_\perp$ is a *CPO* with bottom element, which is called flat.

A *complete lattice* is a poset $(S, \leq)$ such that for every subset X of S there exists $\bigsqcup X$ and $\bigsqcap X$. Let $\top$ denote the *top element* $\bigsqcup S = \bigsqcap \emptyset$ and $\perp$ denote the *bottom element* $\bigsqcap S = \bigsqcup \emptyset$ of S. The elements of a complete lattice can be thought of as points of information and the ordering as an approximation relation between them. Thus, $x \leq y$ means x approximates y (or, x has less or the same information as y) and so $\perp$ is the point of least information. It is easy to check that, for any set S, $\wp(S)$ under the subset ordering $\subseteq$ is a complete lattice, where $\sqcup$ is union, $\sqcap$ is intersection, the top element is S and the bottom element is $\emptyset$. Also $(\wp(S))_\perp$ is a complete lattice.

Given a complete lattice $(L, \leq)$, the set of all partial functions $F = [S \rightharpoonup L]$ inherits the complete lattice structure of L. Let us define $f \preceq g := \forall x \in S. f(x) \leq g(x)$, $(f \sqcup g)(x) := f(x) \sqcup g(x)$, $(f \sqcap g)(x) := f(x) \sqcap g(x)$, $\perp_F := \lambda x \in S. \perp_L$ and $\top_F := \lambda x \in S. \top_L$. In the following $f + g$ will be used to denote $f \sqcup g$.

Continuous and Additive Functions

Let $(L, \leq)$ and $(M, \sqsubseteq)$ be (complete) lattices. A function $f : L \to M$ is *monotonic* if and only if

$$\forall x, y \in L. x \leq y \Longrightarrow f(x) \sqsubseteq f(y).$$

Moreover, f is *continuous* if and only if, for each non-empty chain $D \subseteq L$,

$$f(\bigsqcup_L D) = \bigsqcup_M f(D).$$

Every continuous function is also monotonic, since $x \leq y \Longrightarrow \bigsqcup_M \{f(x), f(y)\} = f(\bigsqcup_L \{x, y\}) = f(y) \Longrightarrow f(x) \sqsubseteq f(y)$.

Complete partial orders correspond to types of data (data that can be used as input or output to a computation) and computable functions are modelled as continuous functions between them.

A partial function $f : S \rightharpoonup S'$ is *additive* if and only if the previous continuity condition is satisfied for each non-empty set. Hence, every additive function is also continuous. Dually, we define *co-continuity* and *co-additivity*, by using $\sqcap$ instead of $\sqcup$.

It can be proved that the composition of monotonic, continuous or additive functions is, respectively, monotonic, continuous or additive.

The mathematical way of expressing that structures are "essentially the same" is through the concept of isomorphism which establishes when structures are isomorphic. A continuous function $f : D \rightarrow E$ between *CPO*s D and E is said to be an *isomorphism* if there is a continuous function $g : E \rightarrow D$ such that $g \circ f = \mathit{Id}_D$ and $f \circ g = \mathit{Id}_E$. Thus f and g are mutual inverses. This is actually an instance of a general definition which applies to a class of objects and functions between them (*CPO*s and continuous functions in this case). It follows from the definition that isomorphic *CPO*s are essentially the same but for a renaming of elements. It can be proved that a function $f : D \rightarrow E$ is an isomorphism if and only if f is bijective and, for all $x, y \in D$, $x \leq_D y \iff f(x) \leq_E f(y)$.

Function Space

Let D, E be *CPO*s. It is a very important fact that the set of all continuous functions from D to E can be made into a complete partial order. The function space $[D \rightarrow E]$ consists of continuous functions $f : D \rightarrow E$ ordered pointwise by $f \sqsubseteq g \iff \forall d \in D.\, f(d) \sqsubseteq g(d)$. This makes the function space a complete partial order. Note that, provided E has a bottom element $\bot_E$, such a function space of *CPO*s has a bottom element, the constantly $\bot_E$ function $\bot_{[D \rightarrow E]} := \lambda d \in D.\, \bot_E$. Least upper bounds of chains of functions are given pointwise, i.e., a chain of functions $f_0 \sqsubseteq f_1 \sqsubseteq \ldots \sqsubseteq f_n \sqsubseteq \ldots$ has lub $\bigsqcup_{[D \rightarrow E]} f_n := \lambda d \in D.\, \bigsqcup_E \{f_n(d)\}_{n \in \mathbb{N}}$.

It is not hard to see that the partial functions $L \rightharpoonup D$ are in 1-1 correspondence with the (total) functions $L \rightarrow D_\bot$, and that, in this case, any total function is continuous; the inclusion order between partial functions corresponds to the "pointwise order" $f \sqsubseteq g \iff \forall \sigma \in L.\, f(\sigma) \sqsubseteq g(\sigma)$ between functions $L \rightarrow D_\bot$. Because partial functions from a *CPO* so does the set of functions $[L \rightarrow D_\bot]$ ordered pointwise. This is the reason why we assumed that, for each set S, $\aleph \subseteq S$, $\aleph \cup S = S$ and $\emptyset \not\subseteq \aleph$.

1.3 Transition Systems

The behavior of systems can be captured by *transition systems*. A *state* is an instantaneous description of the system that captures the values of the variables at a particular instant of time. A *transition* describes the evolution of the system when an action occurs by giving the state before the action occurs and the state after the action occurs. *Computations* of a system can be defined in terms of its transition sequences. A computation is an infinite sequence of states where each state is obtained from the previous state by some transition.

A *Kripke structure* is a kind of state transition graph with a set of states, a set of transitions between states and a function that labels each state with the set of properties which are true in such state. Paths in a Kripke structure model computations of the system.

Definition 1.3.1 (Kripke Structure [CGP99]) *Let* $\mathsf{AP}_{\mathsf{KS}}$ *a set of atomic propositions. We define a* Kripke Structure M_{KS} *over* $\mathsf{AP}_{\mathsf{KS}}$ *as a four tuple* $\mathsf{M}_{\mathsf{KS}} = (\mathsf{S}_{\mathsf{KS}}, \mathsf{S}_{0_{\mathsf{KS}}}, \mathsf{R}_{\mathsf{KS}}, \mathsf{L}_{\mathsf{KS}})$ *where*

1. S_{KS} *is a finite set of states.*
2. $\mathsf{S}_{0_{\mathsf{KS}}} \subseteq \mathsf{S}_{\mathsf{KS}}$ *is the set of initial states.*
3. $\mathsf{R}_{\mathsf{KS}} \subseteq \mathsf{S}_{\mathsf{KS}} \times \mathsf{S}_{\mathsf{KS}}$ *is a transition relation that must be total, that is, for every state* $s \in \mathsf{S}_{\mathsf{KS}}$ *there is a state* $s' \in \mathsf{S}_{\mathsf{KS}}$ *such that* $\mathsf{R}(s, s')$.
4. $\mathsf{L} : \mathsf{S} \rightarrow 2^{\mathsf{AP}_{\mathsf{KS}}}$ *is a function that labels each state with the set of atomic propositions true in that state.*

1.4 Constraint Systems

A *constraint system* is a system of partial information. Formally, a *cylindric constraint system* (with diagonal elements) is defined as a structure $\langle \mathsf{D}, \vdash, \mathcal{V}, \{\exists_x \mid x \in \mathcal{V}\}, \{\delta_{xy} \mid x, y \in \mathcal{V}\}\rangle$ where D is a non-empty set of *tokens*, $\vdash \subseteq \wp(\mathsf{D}) \times \mathsf{D}$ is an *entailment relation*, $\mathcal{V}$ is an infinite set of *variables*, $\exists_x : \mathsf{D} \rightarrow \mathsf{D}$ is the *existential quantifier* or *cylindrification operator*, and δ_{xy} is a *diagonal element*. These components are related each other by several properties. A *constraint* is a subset of D closed by entailment and we use $x = y$ to denote the diagonal element instead of δ_{xy}. The relation $\vdash$ and the operation $\exists_x$ may be extended to constraints. Moreover, we use $\sqcup$ as the least upper bound over constraints.

Since the notion of constraint system is underlying the whole thesis, next we introduce more formally all these notions.

Definition 1.4.1 (Simple constraint system [SRP91]) *Let* D *be a non-empty set of* tokens *or* primitive constraints. *A* simple constraint system *is a structure* $\langle \mathsf{D}, \vdash \rangle$ *where* $\vdash \subseteq \wp_f(\mathsf{D}) \times \mathsf{D}$ *is an* entailment relation *satisfying:*

C1 $u \vdash \mathsf{P}$ *whenever* $\mathsf{P} \in \mathsf{U}$,

C2 $u \vdash \mathsf{Q}$ *whenever* $u \vdash \mathsf{P}$ *for all* $\mathsf{P} \in v$ *and* $v \vdash \mathsf{Q}$,

An element of $\wp_f(\mathsf{D})$ is called a *finite constraint*. We extend $\vdash$ to a relation on $\wp_f(\mathsf{D}) \times \wp_f(\mathsf{D})$ by defining

$$u \vdash v \iff \forall \mathsf{P} \in v,\ u \vdash \mathsf{P} \tag{1.4.1}$$

We also define $u \approx v$ iff $u \vdash v$ and $v \vdash u$.

Definition 1.4.2 (Cylindric constraint system [SRP91]) *We define a cylindric constraint system as a structure* $\langle D, \vdash, \mathcal{V}, \{\exists_x \mid x \in \mathcal{V}\}\rangle$ *such that* $\langle D, \vdash\rangle$ *is a simple constraint system,* $\mathcal{V}$ *is an infinite set of* variables *or* indeterminates *and, for each* $x \in \mathcal{V}$, $\exists_x : \wp_f(D) \to \wp_f(D)$ *is an operation satisfying:*

E1 $u \vdash \exists_x u$,

E2 $u \vdash v$ *implies* $\exists_x u \vdash \exists_x v$,

E3 $\exists_x(u \cup \exists_x v) \approx \exists_x u \cup \exists_x v$,

E4 $\exists_x \exists_y u \approx \exists_y \exists_x u$.

$\exists_x$ *is called the* existential quantifier *or* cylindrification operator.

A set of diagonal elements *for a cylindric constraint system is a family* $\{\delta_{xy} \in D \mid x, y \in \mathcal{V}\}$ *such that*

D1 $\emptyset \vdash \delta_{xx}$,

D2 *if* $y \neq x, z$ *then* $\{\delta_{xz}\} \approx \exists_y\{\delta_{xy}, \delta_{yz}\}$,

D3 *if* $x \neq y$ *then* $\{\delta_{xy}\} \cup \exists_x(u \cup \{\delta_{xy}\}) \vdash u$.

Note that [SRP91] has a couple of typos, since in **D2** the finite constraint $\{\delta_{xz}\}$ is incorrectly replaced by $\{\delta_{xy}\}$. Moreover the condition $y \neq z$ is missing in **D2** and $x \neq y$ is missing in **D3**.

Let Id be a set of process identifiers, $\mathcal{V}$ a denumerable set of variables and $\mathcal{C}$ a (cylindric) constraint system over $\mathcal{V}$. Letters p, x and c are used to range over process identifiers, variables and constraints respectively. With $\vec{x}$ we denote a finite (possibly empty) sequence of variables in $\mathcal{V}$. To simplify the proofs, we assume there is a set $\mathcal{B} \subseteq \mathcal{V}$ denumerable, such that its complement $\mathcal{F}$ is denumerable, too. We call $\mathcal{B}$ the set of *bound variables* and $\mathcal{F}$ the set of *free variables*.

Given a set of variables $V = \{v_1, \ldots, v_n\}$, we denote with $\exists_V c$ the constraint $\exists_{v_1} \cdots \exists_{v_n} c$. Given a constraint c, a variable v is free in c when $\exists_v c \neq c$. Given two sequences $\vec{x} = \{x_i\}_{i \leq n}$ and $\vec{y} = \{y_i\}_{i \leq n}$ of variables, both of length n, we denote with $\vec{x} = \vec{y}$ the constraint $\sqcup_{i \leq n}(x_i = y_i)$.

Definition 1.4.3 (Free variables) *Let* $\mathcal{C}$ *be a constraint system over* $\mathcal{V}$ *and* c *a constraint of* $\mathcal{C}$. *A variable* v *is* (essentially) free *in* c *when* $\exists_x c \neq c$. *We denote with* $fv(c)$ *the set of free variables in* c.

The set of free variables in c is sometimes called the *dimension set* of c, according to [HMT71].

In Proposition 1.4.4 we show some properties that are satisfied by free variables. Those properties are used to prove the results presented in the first part of the dissertation.

Proposition 1.4.4 *Free variables satisfy the following conditions:*

1. $fv(c \sqcup d) \subseteq fv(c) \cup fv(d)$;

2. $fv(\exists_x c) \subseteq fv(c) \setminus \{x\}$

Proof. We prove the points separately.

Point 1 We show that, if a variable is not in the second set, then it cannot be in the first one. In other terms, if $v \notin fv(c) \cup fv(d)$ then it is true that $\exists_v c = c$ and $\exists_v d = d$, and therefore $\exists_v (c \sqcup d) = c \sqcup d$, i.e., $v \notin fv(c \sqcup d)$.

Point 2 We know that $x \notin fv(\exists_x.c)$ since $\exists_x \exists_x c = \exists_x c$. Moreover, if $v \notin fv(c)$ then $\exists_v \exists_x c = \exists_x \exists_v c = \exists_x c$, i.e., $v \notin fv(\exists_x c)$.

■

Note that the equality does not hold in general. For example $fv(x = y) = \{x, y\}$ but $fv(\exists_y.x = y) = \emptyset$.

Sometimes finite constraints are not enough, since we need to deal with the continuous accumulation of simple constraints.

Definition 1.4.5 (Elements [SRP91]) *We define an* element c *of a constraint system* $\langle D, \vdash \rangle$ *as a subset of* D *closed by entailment, i.e., such that* $u \subseteq_f c$ *and* $u \vdash P$ *implies* $P \in c$. *With an abuse of notation, we denote by* $|D|$ *the set of elements of the constraint system* $\langle D, \vdash \rangle$.

Entailment and cylindrification may be extended to elements of a constraint system, according to the following definitions:

$$c \vdash d \iff c \supseteq d \tag{1.4.2}$$
$$\exists_x c = \{P \mid \exists_x u \vdash P, \text{ for some } u \subseteq_f c\} \tag{1.4.3}$$

1.4.1 Cylindric semilattices

Actually, the definition of constraint system is restrictive in order to force decidability or semi-decidability of certain operations. *Cylindric semilattices* are a more general notion. Finite constraints and elements of a constraint system form two different cylindric semilattices.

Definition 1.4.6 *A* cylindric semilattice *is a structure* $\langle C, \sqsubseteq, \bot, \mathcal{V}, \{\exists_x \mid x \in \mathcal{V}\}, \{\delta_{xy} \mid x, y \in \mathcal{V}\}\rangle$ *such that* $\langle C, \sqsubseteq \rangle$ *is an upper semilattice,* $\bot$ *is the least element of* C, $\mathcal{V}$ *is a set of variables,* $\exists_x : C \to C$ *and* $\delta_{xy} \in C$. *These are subject to the following conditions:*

E1 $c \sqsupseteq \exists_x c$,

E2 $\exists_x$ *is monotonic,*

E3 $\exists_x(c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$*,*

E4 $\exists_x \exists_y c = \exists_y \exists_x c$*,*

D1 $\delta_{xx} = \perp$*,*

D2 *if* $y \neq x, z$ *then* $\delta_{xz} = \exists_y(\delta_{xy} \sqcup \delta_{yz})$*,*

D3 *if* $x \neq y$ *then* $\delta_{xy} \sqcup \exists_x(c \sqcup \delta_{xy}) \sqsupseteq c$.

Our definition is almost the same as the one given for *cylindric algebras* in [HMT71], but we use upper semilattices instead of boolean algebras for the underlying partially ordered set C.

Note that condition **E2** for cylindric semilattices is redundant. Actually, if $c \sqsubseteq d$ then $\exists_x c \sqsubseteq d$ and therefore $\exists_x c \sqcup d = d$. By **E3**

$$\exists_x d = \exists_x(\exists_x c \sqcup d) = \exists_x c \sqcup \exists_x d$$

which means that $\exists_x c \sqsubseteq \exists_x d$.

Note that the same does not hold in the set of axioms for constraint systems. Actually, consider a first order language $\mathcal{L}$ with equality and two predicate symbols p and q of arity one. Let $\mathcal{V}$ be a denumerable set of variables and α a variable in $\mathcal{V}$. We denote with D the following set of formulas:

$$D = \{x = y \mid x, y \in \mathcal{V}\} \cup \{p(x), q(x) \mid x \in \mathcal{V}\} \cup \{\exists_\alpha p(\alpha)\}$$

We denote with $\vdash_\mathcal{L}$ the entailment in the logic and with $\vdash^*_\mathcal{L}$ the entailment under the hypothesis $\forall x.p(x) \leftrightarrow q(x)$.

Our purpose is to build a constraint system with set of tokens D. Given $u = \{\phi_1, \ldots, \phi_n\} \in \wp_f(D)$, we define $u \vdash \phi$ iff $\phi_1 \wedge \cdots \wedge \phi_n \vdash^*_\mathcal{L} \phi$. Moreover, we define

$$\exists_x u = \{\phi \in D \mid \exists_x(\phi_1 \wedge \cdots \wedge \phi_n) \vdash_\mathcal{L} \phi\} \setminus \{x = x \mid x \in \mathcal{V}\}$$

It is possible to check that, if we identify δ_{xy} with the formula $x = y$, all the conditions for a cylindric constraint systems are satisfies, with the only exception of **E3**. Actually, if $u = \{q(x)\}$ and $v = \{q(x), p(x)\}$, we have $\exists_x u = \emptyset$ and $\exists_x v = \{r\}$. Therefore $u \vdash v$ but $\exists_x u \not\vdash \exists_x v$.

Given a constraint system with diagonal elements $\mathcal{C} = \langle D, \vdash, \mathcal{V}, \{\exists_x \mid x \in \mathcal{V}\}, \{\delta_{xy} \mid x, y \in \mathcal{V}\}\rangle$, we have two obvious examples of cylindric semilattices: finite constraints and elements.

Proposition 1.4.7 *Let* $D = \wp_f(C)/_\approx$ *the equivalence class of finite constraints w.r.t. entailment and we define* $[u] \sqsupseteq [v]$ *iff* $u \vdash v$. *It is the case that* D *is an upper semilattice with* $[u] \sqcup [v] = [u \cup v]$ *and bottom element* $[\emptyset]$. *The operation* $\exists_x$ *is immediately extended to* D *by taking* $\exists[u] = [\exists u]$*, which can be proved to be well defined. Then* $\langle D, \sqsubseteq, [\emptyset], \mathcal{V}, \{\exists_x \mid x \in \mathcal{V}\}, \{[\delta_{xy}] \mid x, y \in \mathcal{V}\}\rangle$ *is a cylindric semilattice.*

The proof of this proposition is simple and only involves elementary operations with equivalence classes of finite constraints.

Proposition 1.4.8 *For each* $X \subseteq D$, *we denote with* $\bar{X}$ *the closure of* X, *i.e., the set* $\bar{X} = \{P \mid \exists u \subseteq_f X.\ u \vdash P\}$. *Then* $(|D|, \subseteq)$ *is an upper semilattice with* $c \sqcup d = \overline{c \cup d}$ *and bottom element* $\overline{\emptyset}$. *The structure* $\langle |D|, \subseteq, \overline{\emptyset}, \{\exists_x \mid x \in \mathcal{V}\}, \{\overline{\{\delta_{xy}\}} \mid x, y \in \mathcal{V}\rangle$ *is a cylindric semilattice.*

Proof. The fact that $(|D|, \subseteq)$ is an upper semilattice with given lowest upper bound and least element is trivial. To prove the conditions regarding cylindrification and diagonal elements, we first observe that, given $u \in \wp_f(D)$ and $c, d \in |D|$, the following property holds:

$$u \subseteq_f c \sqcup d \iff \text{there exist } v \subseteq_f c \text{ and } w \subseteq_f d \text{ such that } v \cup w \vdash u$$

Now, consider the proof of the condition **E3**. By the definition of $\exists_x$, $P \in \exists_x(c \sqcup \exists_x d)$ iff there is $u \subseteq_f c \sqcup \exists_x d$ such that $\exists_x u \vdash P$. By the previous observation, $u \subseteq_f c \sqcup \exists_x d$ iff there are $v_1 \subseteq_f c$ and $v_2 \subseteq_f \exists_x d$ such that $v_1 \cup v_2 \vdash u$. Finally, again by definition of $\exists_x$ on elements, $v_2 \subseteq_f \exists_x d$ iff there is $w \subseteq_f d$ such that $\exists_x w \vdash v_2$. In one formula:

$$\begin{aligned} P \in \exists_x(c \sqcup \exists_x d) \iff & \exists u, v_1, v_2, w \in \wp_f(D) \\ & v_1 \subseteq_f c, w \subseteq_f d, \exists_x w \vdash v_2, v_1 \cup v_2 \vdash u, \exists_x u \vdash P \end{aligned}$$

By **E2** of constraint systems, this is equivalent to

$$P \in \exists_x(c \sqcup \exists_x d) \iff \exists v_1, w \in \wp_f(D).\ \exists_x(v_1 \cup \exists_x w) \vdash P$$

In the same way $P \in \exists_x c \sqcup \exists_x d$ iff there are $v_1 \subseteq_f c$ and $w \subseteq_f d$ such that $\exists_x v_1 \cup \exists_x w \vdash P$. By definition **E3** for constraint systems, these are equivalent.

Now, let us consider the proof for the condition **D2**. Note that $u \subseteq \overline{\{\delta_{xy}\}}$ iff $\{\delta_{xy}\} \vdash u$. Therefore, by following the same idea as above

$$P \in \exists_y(\overline{\{\delta_{xy}\}} \sqcup \overline{\{\delta_{yz}\}}) \iff \exists_x(\{\delta_{xy}\} \cup \{\delta_{yz}\}) \vdash P \iff \exists_x(\delta_{xy}, \delta_{yz}) \vdash P$$

By **D2** for constraint systems, $P \in \exists_y(\overline{\{\delta_{xy}\}} \sqcup \overline{\{\delta_{yz}\}}) \iff P \in \overline{\{\delta_{xz}\}}$.

By following the same line, all the conditions required by the definition of cylindric semilattice may be proved to hold. ■

We conclude this digression on constraint systems by proving that there is a precise relation between the two cylindric semilattices considered so far.

Proposition 1.4.9 *The map* $^- : D/_{\approx} \to |D|$ *given by* $\overline{[u]} = \bar{u}$ *is an embedding (i.e., an injective homomorphism) from the algebra of finite constraints to the algebra of elements.*

Proof. The proof is simple. We only show that $^-$ is an homomorphism for $\exists_x$. We need to prove that

$$\overline{\exists_x(u \cup v)} = \exists_x(\bar{u} \sqcup \bar{v})$$

Following the same line of the previous theorem, we obtain that $P \in \exists_x(\bar{u} \sqcup \bar{v})$ iff $\exists_x(u \cup v) \vdash P$ which is equivalent to $P \in \overline{\exists_x(u \cup v)}$. ■

1.4.2 Substitutions

Following the idea presented in [HMT71] for cylindric algebras, a *substitution* is a map $f : \mathcal{V} \to \mathcal{V}$ such that $dom(f) = \{v \in \mathcal{V} \mid f(v) \neq v\}$ is finite. A *renaming* is a bijective substitution. We denote a substitution f with the set $\{v_1/f(v_1), \ldots, v_n/f(v_n)\}$ such that $\{v_1, \ldots, v_n\} = dom(f)$. If $\vec{x} = x_1, \ldots, x_n$ and $\vec{y} = y_1, \ldots, y_n$, then $\{\vec{x}/\vec{y}\}$ is the substitution which maps each x_i with the corresponding y_i.

Definition 1.4.10 (Substitution Operator) *Let c be a constraint. We denote with $c[f]$ a new constraint obtained by simultaneously replacing each copy of $v \in dom(f)$ with the corresponding $f(v)$. Formally, we may define*

$$c[\vec{x}/\vec{y}] = \exists_{\vec{\alpha}}(\vec{y} = \vec{\alpha} \sqcup \exists_{\vec{x}}.(\vec{\alpha} = \vec{x} \sqcup c)) \tag{1.4.4}$$

where $\vec{\alpha}$ is a sequence of variables, of the same length of $\vec{x}$ and $\vec{y}$, such that $(fv(c) \cup \vec{y} \cup \vec{x}) \cap \vec{\alpha} = \emptyset$.

Note that a sequence $\vec{\alpha}$ which satisfies this condition does not always exists, since it is possible to have a constraint c such that $fv(c) = \mathcal{V}$.

We say that c is *domain complemented* when $\mathcal{V} \setminus fv(c)$ is infinite. If c is domain complemented, then $c[f]$ does exists for every substitution f, and the definition does not depend from the choice of the sequence of fresh variables $\vec{\alpha}$.A cylindric lattice is said to be *dimension complemented* when, for each element c, $fv(c) \neq \mathcal{V}$. For a dimension complemented cylindric lattice, it is possible to build an analogous of parallel substitution.

In Proposition 1.4.11 we show some properties that are satisfied by substitutions. Again those properties are used to prove some results presented in this thesis.

Proposition 1.4.11 *These are some of the properties of substitutions:*

1. $c[\vec{x}/\vec{y}]$ *is well defined;*

2. *if* $c \sqsubseteq d$ *then* $c[f] \sqsubseteq d[f]$*;*

3. *if* $f' = f|_{fv(c)}$ *then* $c[f] = c[f']$*;*

4. $c[f_1][f_2] = c[f_1 \circ f_2]$*;*

5. *if* f *is a renaming, then* $\exists_{f(\vec{x})}c[f] = (\exists_{\vec{x}}c)[f]$.

Proof. We prove the several points separately.

Point 1 Following (1.4.4), we have to prove that

$$c_1 = \exists_{\vec{\alpha}}(\vec{y} = \vec{\alpha} \sqcup \exists_{\vec{x}}.(\vec{\alpha} = \vec{x} \sqcup c)) \sqsubseteq \exists_{\vec{\beta}}(\vec{y} = \vec{\beta} \sqcup \exists_{\vec{x}}.(\vec{\beta} = \vec{x} \sqcup c)) = c_2$$

for every choice of $\vec{\alpha}$ and $\vec{\beta}$. It is enough to prove the property when α and β are disjoint. Moreover, note that $fv(c_1) \subseteq fv(c) \cup \vec{y}$ is disjoint from $\vec{\beta}$ and therefore it is enough to prove

$$c_1 \sqsubseteq \vec{y} = \vec{\beta} \sqcup \exists_{\vec{x}}.(\vec{\beta} = \vec{x} \sqcup c)$$

By using rule **D2** of constraint systems (see Definition 1.4.2,

$$\vec{y} = \vec{\beta} \sqcup \exists_{\vec{x}}.(\vec{\beta} = \vec{x} \sqcup c) = \exists_{\vec{\alpha}}(\vec{y} = \vec{\alpha} \sqcup \vec{\alpha} = \vec{\beta} \sqcup \exists_{\vec{x}}.(\vec{\beta} = \vec{x} \sqcup c))$$

and moreover

$$\exists_{\vec{\alpha}}(\vec{y} = \vec{\alpha} \sqcup \vec{\alpha} = \vec{\beta} \sqcup \exists_{\vec{x}}.(\vec{\beta} = \vec{x} \sqcup c)) = \\ \exists_{\vec{\alpha}}(\vec{y} = \vec{\alpha} \sqcup \vec{\alpha} = \vec{\beta} \sqcup \exists_{\vec{x}}.(\vec{\alpha} = \vec{\beta} \sqcup \vec{\beta} = \vec{x} \sqcup c)) \sqsupseteq c_1 \quad (1.4.5)$$

which proves the theorem.

Point 2 The second property follows from the monotonicity of $\sqcup$ and $\exists$, by choosing the same sequence α of fresh variables for both the existential quantifiers.

Point 3 To prove the third property it is enough to show that given $f = \{x/y, \vec{v}/\vec{w}\}$ with $x \notin fv(c)$ then $c[f] = c[\vec{v}/\vec{w}]$. We try to construct the equation for this case following (1.4.4). If $x \notin fv(c)$ then $\exists_x c = c$ and therefore we can say that $\exists_{\vec{v}}(\exists_x(\alpha = x \sqcup \vec{\beta} = \vec{v} \sqcup c)) = \exists_{\vec{v}}((\exists_x \alpha = x) \sqcup \vec{\beta} = \vec{v} \sqcup c) = \exists_{\vec{v}}(\vec{\beta} = \vec{v} \sqcup c) = c'$. Then we can see that this c' denotes the last part of the equation for $c[f]$. Moreover, we see that $\alpha \notin fv(c')$, and therefore $c[f] = \exists_{\vec{\beta}}\exists_\alpha(y = \alpha \sqcup \vec{w} = \vec{\beta} \sqcup c') = \exists_{\vec{\beta}}(\vec{\beta} = \vec{w} \sqcup c') = c[\vec{v}/\vec{w}]$ which proves the theorem.

Point 4 For the proof of the fourth case, we assume $f_1 = \{\vec{x}/\vec{y}\}$ and $f_2 = \{\vec{v}/\vec{w}\}$ and following the same (1.4.4) we obtain

$$c[f_1][f_2] = \exists_{\vec{\beta}}(\vec{\beta} = \vec{w} \sqcup \exists_{\vec{v}}(\vec{\beta} = \vec{v} \sqcup \exists_{\vec{\alpha}}(\vec{y} = \vec{\alpha} \sqcup \exists_{\vec{x}}(\vec{\alpha} = \vec{x} \sqcup c))))$$

and note that $f_1 \circ f_2 = \{\vec{x}/f_2(\vec{y})\} \uplus \{v/f_2(v) \mid v \in dom(f_2) \setminus dom(f_1)\} = \{\vec{z}/\vec{t}\}$. If we choose $\vec{\alpha}$ and $\vec{\beta}$ disjoint, we may move $\exists_{\vec{\alpha}}$ outwards obtaining

$$c[f_1][f_2] = \exists_{\vec{\alpha},\vec{\beta}}(\vec{\beta} = \vec{w} \sqcup \exists_{\vec{v}}(\vec{\beta} = \vec{v} \sqcup \vec{y} = \vec{\alpha} \sqcup \exists_{\vec{x}}(\vec{\alpha} = \vec{x} \sqcup c)))$$

We can distinguish $\vec{y}$ in the two subsets $\vec{y}_1$ and $\vec{y}_2$ being $\vec{y}_1$ the set of variables which are also in $\vec{v}$ and $\vec{y}_2 = \vec{y} \setminus \vec{y}_1$. Then, since variables in $\vec{y}_2$ are not in the domain of f_2, then we can put them outside the scope of such quantification obtaining

$$c[f_1][f_2] = \exists_{\vec{\alpha},\vec{\beta}}(\vec{\beta} = \vec{w} \sqcup \vec{y}_2 = \vec{\alpha}_2 \sqcup \exists_{\vec{v}}(\vec{\beta} = \vec{v} \sqcup \vec{y}_1 = \vec{\alpha}_1 \sqcup \exists_{\vec{x}}(\vec{\alpha} = \vec{x} \sqcup c)))$$

Note that, $\vec{\beta} \uplus \vec{y}_2$ corresponds to $\vec{t}$. Now we can also distinguish $\vec{x}$ in the two subsets $\vec{x}_1$ and $\vec{x}_2$ being $\vec{x}_1$ the variables which are shared with $\vec{v}$. Then we have

$$c[f_1][f_2] = \exists_{\vec{\alpha},\vec{\beta}}(\vec{\beta} = \vec{w} \sqcup \vec{y}_2 = \vec{\alpha}_2 \sqcup \exists_{\vec{v},\vec{x}_2}(\vec{\beta} = \vec{v} \sqcup \vec{y}_1 = \vec{\alpha}_1 \sqcup \vec{\alpha}_2 = \vec{x}_2 \sqcup c))$$

which coincides with the definition of substitution for $c[f_1 \circ f_2]$.

Point 5 Since $\mathsf{c} = \mathsf{c}[\mathsf{f}][\mathsf{f}^{-1}]$, for this last property it is enough to prove half of the equality, namely that $\exists_{\vec{y}}\mathsf{c}[\vec{x}/\vec{y}] \sqsupseteq \exists_{\vec{x}}\mathsf{c}$. Note that since f is a renaming, $\vec{x}$ and $\vec{y}$ are equals when viewed as sets, therefore $\vec{y} \cap \mathit{fv}(\exists_{\mathsf{x}}\mathsf{c}) = \emptyset$, and it is enough to prove $\mathsf{c}[\vec{x}/\vec{y}] \sqsupseteq \exists_{\vec{x}}\mathsf{c}$. It is immediate to check that $\mathsf{c}[\vec{x}/\vec{y}] \sqsupseteq \exists_{\vec{y}}\exists_{\vec{x}}\mathsf{c} = \mathsf{c}$, and this proves the theorem.

■

Actually, all these properties make this notion of substitution the counterpart of standard syntactic substitution for first order formulas. However, we have a problem. If $\mathcal{C}$ is a constraint system, while the cylindric semilattice of finite constraint is domain complemented, the same does not hold for the cylindric semilattice of elements.

In the semantics of cc and related languages, we may overcome this problem by partitioning the set $\mathcal{V}$ of variables into two infinite sets $\mathcal{F}$ of free variables and a set $\mathcal{B}$ of bound variables. Constraints in programs and environments are only allowed to take free variables in $\mathcal{F}$, so that substitution may be defined. Obviously, we must be careful that semantic definitions do not produce unsafe constraint with some free variable in $\mathcal{B}$.

1.5 Closure Operators

Let $(\mathsf{C}, \leq)$ be a complete lattice, a *partial Moore family* in C is a set $\mathsf{S} \subseteq \mathsf{C}$ which is meet closed for non-empty set, i.e., such that if $\emptyset \neq \mathsf{X} \subseteq \mathsf{S}$, then $\bigwedge \mathsf{C} \in \mathsf{S}$. A *partial closure operator* in C is a partial map $\rho : \mathsf{C} \rightharpoonup \mathsf{C}$ such that, if $\mathsf{c} \in \mathsf{C}$ and $\rho(\mathsf{c})$ is defined, then:

- $\rho(\mathsf{c}) \sqsupseteq \mathsf{c}$ (ρ is *extensive*);
- $\rho(\rho(\mathsf{c})) = \rho(\mathsf{c})$ (ρ is *idempotent*);
- if $\mathsf{c}' \sqsubseteq \mathsf{c}$, $\rho(\mathsf{c}') \sqsubseteq \rho(\mathsf{c})$, ($\rho$ is *monotone*).

With every partial Moore family $\mathsf{S} \subseteq \mathsf{C}$, we may associate a partial closure operator $\bar{\mathsf{S}} : \mathsf{C} \rightharpoonup \mathsf{C}$ as follows:

$$\bar{\mathsf{S}}(\mathsf{c}) = \begin{cases} \bigwedge\{\mathsf{c}' \in \mathsf{C} \mid \mathsf{c}' \geq \mathsf{c}\} & \text{if } \{\mathsf{c}' \in \mathsf{C} \mid \mathsf{c}' \geq \mathsf{c}\} \neq \emptyset, \\ \aleph & \text{otherwise.} \end{cases} \tag{1.5.1}$$

On the contrary, for each partial closure operator $\rho : \mathsf{C} \rightharpoonup \mathsf{C}$, we associate a partial Moore family given by the set of fixpoints of ρ. These two transformations are one the inverse of the other, and define an isomorphism between partial closure operators ordered pointwise and partial Moore families ordered by containment.

A *Moore family* in C is a partial Moore family which contains the greatest element in C, and a closure operator is a partial closure operator which is defined for every element of C. The correspondence viewed above restricts naturally to Moore families and closure operators.

3

The tcc language

Concurrent constraint programming was first introduced by Vijay Saraswat in 1989 [Sar89] as a simple and powerful model of concurrent systems (see also [Sar93]). In this model the concept of *store as valuation* of von Neumann is replaced by the notion of *store as constraint*: constraints are accumulated in a store providing partial information about the possible values that variables can take. The formal definition of this programming paradigm can be found in [SR90, SRP91, Sar93].

In this chapter we describe a language which was presented a few years later by Saraswat, Jagadeesan and Gupta as an evolution of the cc model: the *timed concurrent constraint programming* (tcc) language. tcc was defined in [SJG94a, SJG94b] basically by adding the notion of time to the model. The fundamental contribution of the tcc model is to augment the ability of constraint programming to detect *positive information* with the ability to detect *negative information*, i.e., to detect the absence of some signal or information. Such negative information is crucial to model reactive and real-time computations (see [HMP92]).

In particular, tcc extends the deterministic fragment of the cc paradigm with agents that are able to model the temporal behavior and also notions such as *timeout* and *preemption*. The notion of timeout is defined as the ability to wait for a specific signal and, if a limit of time is reached and such signal is not present, then an exception program is executed. Another interesting behavior of reactive sytems is preemption which is defined as the ability to abort a process when a specific signal is detected. Actually, in the case of tcc it is captured the notion of weak preemption, i.e., it is not possible to abort a process in the same time instant when the signal is detected but in the following one. In the more refined model presented in [SJG96] it was introduced an appropriate framework to model the strong preemption behavior.

It is important to remark that timed concurrent constraint programming was developed as a simple model for determinate, timed, and reactive systems. Using this declarative model the typical advantages of the declarative paradigm are gained. For example, programming in this model is more intuitive and reasoning with the derived languages is easier than with imperative languages. Another important property is that the specification is executable, that is what you prove is what you execute. Moreover, the language supports hierarchical and modular construction of specifications (programs). Finally, construction and analysis of programs and specifications is easier since this is a deterministic language.

In the next sections we present the syntax and the semantics of the language. We also discuss the main characteristics of the language.

3.1 Syntax

The original syntax of tcc given in [SJG94a] is presented in Figure 3.1. If the reader is familiar with the cc framework it is easy to see that we can divide the constructs[1] of the language into two categories. The first category corresponds to the agents inherited from the deterministic fragment of cc. We call such constructs *concurrent constraint* (cc) constructs. The second category is composed of those constructs that cause "extension over time" and we call these constructs *timing* constructs.

(Agents)	A ::=	c	– Tell
	$\mid$	now c then A	– Timed Positive Ask
	$\mid$	now c else A	– Timed Negative Ask
	$\mid$	next A	– Unit Delay
	$\mid$	abort	– Abort
	$\mid$	skip	– Skip
	$\mid$	$A \parallel A$	– Parallel composition
	$\mid$	X^A	– Hiding
	$\mid$	g	– Procedure call
(Procedure Calls)	g ::	$p(t_1, \ldots, t_n)$	
(Declarations)	D ::	g :: A	– Definition
	$\mid$	D.D	– Conjunction
(Programs)	P ::	$\{D.A\}$	

Figure 3.1: Syntax for tcc programs.

Intuitively, the execution of a tcc program evolves by steps: in each step it is accumulated all the information that can be generated by the cc constructs. When no more information can be generated by the cc constructs we say that it has been reached a *resting point* (or *quiescent point*). Thus, when a resting point is reached the execution of the program moves to the following time instant by executing the timing constructs. During that passage from one time instant to the following one, all the information accumulated since that moment is removed. Then the process is repeated by starting a new computation of the cc constructs generated by the timing constructs with the empty store.

Tell, *Parallel Composition* and *Timed Positive Ask* agents are cc constructs. We say that they are instantaneous agents, i.e., agents which do not cause extension

[1] In this thesis we use equivalently the terms "construct", "agent", and "operator".

over time. In particular, Parallel Composition is the explicit representation of concurrency in the language, the Timed Positive Ask agent allows the synchronization of processes and the Tell agent represents the state evolution of the system.

The second class of constructs are timing constructs. *Timed Negative Ask*, *Unit Delay* and *Abortion* cause extension over time. The Unit Delay agent forces a process to start in the next time instant. The Timed Negative Ask agent is a conditional version of Unit Delay that allows the system to detect negative information: it causes a process to be started in the next time instant if, on the quiescence of the current time instant, the store was not strong enough to entail some specific information. This means that the absence of information can cause the execution of some agent which is a crucial point in the modeling of reactive systems. Finally, the Abortion agent terminates the execution of all the processes in the next time instant.

Now we extend the definition of free variables given in Definition 1.4.3 to agents. Given a tcc agent A, we denote with $fv(A)$ the set of free variables in A, i.e., those variables x which occur in A outside the scope of any hiding operator relative to x. Formally we have:

$$\begin{aligned}
fv(\mathsf{c}) &\text{ as in Definition 1.4.3}\\
fv(A_1 \parallel A_2) &= fv(A_1) \cup fv(A_2)\\
fv(\mathsf{skip}) &= \emptyset\\
fv(\mathsf{abort}) &= \emptyset\\
fv(\mathsf{now}\ \mathsf{c}\ \mathsf{else}\ A) &= fv(A) \cup fv(\mathsf{c})\\
fv(\mathsf{now}\ \mathsf{c}\ \mathsf{then}\ A) &= fv(A) \cup fv(\mathsf{c})\\
fv(\mathsf{next}\, A) &= fv(A)\\
fv(x\hat{}A) &= fv(A) \setminus \{x\}
\end{aligned}$$

With $vars(A)$ we denote the set of variables in an agent A. With $bv(A)$ we denote the set of bound variables in A, i.e., the set of all the variables x such that there is a subagent $x\hat{}B$ in A. We say that an agent A is *well formed* when $fv(A) \subseteq \mathcal{F}$, $bv(A) \subseteq \mathcal{B}$ and there are no two different hiding operators for the same variable. In the following, we only consider well formed agents.

A *program* is a pair $D.A$ where D is a set of definitions and A is an agent. A *definition* is written as $p(\vec{x}) :: A_p$ where $p \in \mathsf{Id}$, A_p is an agent and $fv(A_p) \subseteq \vec{x}$. We make two assumptions over programs:

1. in each program, for each process identifier there is only one different definition, and

2. every occurrence of an agent $t(\vec{y})$ inside a definition $p(\vec{x}) :: A_p$ is within the scope of a $\mathsf{next}\ A$ or $\mathsf{now}\ \mathsf{c}\ \mathsf{else}\ A$ construct.

We only consider well formed programs. Note that the restriction to $fv(A_p) \subseteq \vec{x}$ is important if we want the correspondence between operational and denotational semantics to hold. This is due to the fact that the operational semantics follows the rule of static scoping, while the denotational semantics follows the rule of dynamic scoping (already pointed out in [NPV02a]). The easiest way to solve the problem is to apply the above said restriction about the free variables which occur in a definition.

The restriction concerning the occurrences of agents of the form $\mathsf{t}(\vec{y})$ is needed to avoid recursion within a fixed time instant. In this way, the behavior of the program in a time instant is simple enough to be described by finite state automata.

Given $\vec{x} = x_1 \ldots x_n$, $\vec{y} = y_1 \ldots y_n$, if f is the substitution $\{\vec{x}/\vec{y}\}$, we define the new agent $A[f]$ obtained by applying f to all the constraints c occurring in A and by replacing each x_i with the corresponding y_i in the procedure calls and hiding operators. Note that, at this level, we do not require $A[f]$ to preserve bound variables, and therefore we have that $(x\hat{\ }p(x,y))[x/z] = z\hat{\ }p(z,y)$. Quite often, we do not want to distinguish between agents which only differs for the name of bound variables. Therefore, we define the relation $\sim_p$ of α-conversion as the smallest congruence w.r.t. the syntactic operators that define agents such that

$$A[x/z] \sim_p A'[x'/z] \land z \neq x, x' \land z \notin \mathit{vars}(A) \cup \mathit{vars}(A') \Longrightarrow x\hat{\ }A \sim_p x'\hat{\ }A'$$

Then, we can apply a substitution f to the class $[A]_{\sim_p}$ as $[A]_{\sim_p}[f] = B[f]$ where $B \sim_p A$ and $\mathit{bv}(B) \cap \mathit{vars}(f) = \emptyset$. Given a sequence of variables $\vec{y}$, we write $p(\vec{x}) :: A_p \ll_{\vec{y}} D$ when $p(\vec{x}) :: A'_p \in D$, $A_p \sim_p A'_p$ and $\mathit{bv}(A'_p) \cap (\vec{x} \cup \vec{y}) = \emptyset$.

It is important to remark some questions about the definition of the tcc language. Note that each occurrence of a recursive call must be preceded by a timing construct. In other words, the recursion within a time instant is not allowed. This restriction provide the authors the possibility to ensure that the time necessary to reach a quiescence point is finite. Thus, it is true that a resting point is always reached.

Moreover, in [SJG94a, SJG96] the definition of other possible operators based in the constructs presented in Figure 3.1 can be found. Those constructs are typically used in other frameworks (such as ESTEREL [Ber00], LUSTRE [HCRP91] or SIGNAL [GGBM91]) whose scope is similar to the one of tcc language: to model reactive systems. For example, it can be defined the always A construct which execute the agent A at each time instant. Other examples are the now c then A else B, *multiple prioritized waits*, whenever c do A or do A watching c agents. All these constructs are defined using the operators showed in Figure 3.1. Thus we could always transform a complex program that uses these new operators into a simpler one which uses only basic operators.

3.2 Operational Semantics

The operational semantics of the language was defined in [SJG94a] by extending the operational semantics of cc. The semantics were presented as *two* transition relations defined over *configurations*. A configuration was defined as a multiset of agents. In this section we show the operational semantics of the language in a bit different way. In this thesis a configuration is defined as a set of variables *and* a multiset of agents instead of a single multiset of agents.

This is because [SJG94a] does not consider the problem of defining a suitable notion of observable behavior, problem which is only addressed in [SJG94b] where the set of new introduced variables is implicitly calculated from a derivation. The two operational semantics are equivalent but we prefer this new notation which is cleaner and simplifies the proofs of the results presented in Chapter 4. Actually, this new notation for the operational semantics is inspired in the operational semantics given in [NPV02b] for the ntcc language, which can be seen as an evolution of tcc.

The idea is that given a program specification $D.A$, the initial configuration is defined as $\langle \emptyset, A \rangle$. Then all the cc operators are executed following the transition relation $\longrightarrow_D$ defined in Figure 3.2 (i.e., agents behave according to the transition relation $\longrightarrow_D$ without passing of time). Due to the syntactic restrictions on the agents, all the $\longrightarrow_D$-derivations from a configuration $\langle V, \Gamma \rangle$ are finite, so that we eventually reach a resting point. Once the resting point is reached no further information may be adjoined in the current time instant, thus we can deduce the negative information. At that point we move to the following time instant by executing the timing constructs in the configuration. All the information obtained in the previous time instant is removed. Then the process is repeated: the cc constructs are executed to reach the following resting point.

Next we describe formally all these intuitions. The operational semantics are defined as the minimal relation generated by the transition rules showed in Figure 3.2. Note that due to space problems the denominator of the last rule is showed in two different lines.

Thus, from now we define a configuration Ω as a pair $\langle V, \Gamma \rangle$ where V is a finite set of variables and Γ is a multiset of agents. Intuitively, the set V contains those variables in $\mathcal{V}$ which are local to the agents in Γ and should not be affected by the environment. given a multiset of agents Γ, $\sigma(\Gamma)$ is defined as the sub-multiset of constraints in Γ called the *store*. Moreover, with $\uplus$ we denote multiset union and we also use $\Gamma \uplus A$ as a short form for $\Gamma \uplus \{\{A\}\}$.

In Figure 3.2 you can see two different transition relations: the first one ($\longrightarrow_D$) represents the transitions *within* a time instant whereas the second transition relation ($\rightsquigarrow_D$) represents the pass from one time instant to the following one. Moreover, in the definition of rule $R_{\rightsquigarrow_D}$, Δ ranges over multisets of agents that do not contain timing agents.

Given a transition $\langle V, \Gamma \rangle \longrightarrow_D \langle V', \Gamma' \rangle$, the agent in the left-hand side which is rewritten is called the *principal* agent. An agent A in $\langle V, \Gamma \rangle$ is *active* when there is an outgoing transition such that A is the principal agent. A configuration $\langle V, \Gamma \rangle$ without outgoing $\longrightarrow_D$-transitions is called *resting point* (or quiescent point).

The assumption that agents are well formed plays a role in R_{def}, where it ensures

Given a set of definitions D, we define the transition systems $\rightarrow_{\mathrm{D}}$ and $\rightsquigarrow_{\mathrm{D}}$ over configurations with the following rules:

$$
\begin{aligned}
R_{\mathsf{skip}} &: \langle V, \Gamma \uplus \mathsf{skip}\rangle \rightarrow_{\mathrm{D}} \langle V, \Gamma\rangle \\
R_{\mathsf{abort}} &: \langle V, \Gamma \uplus \mathsf{abort}\rangle \rightarrow_{\mathrm{D}} \langle V, \mathsf{abort}\rangle \\
R_{\parallel} &: \langle V, \Gamma \uplus A_1 \parallel A_2\rangle \rightarrow_{\mathrm{D}} \langle V, \Gamma \uplus A_1 \uplus A_2\rangle \\
R_{\widehat{\ }} &: \langle V, \Gamma \uplus x\widehat{\ }A\rangle \rightarrow_{\mathrm{D}} \langle V \cup \{y\}, \Gamma \uplus A[x/y]\rangle \text{ where } y \in \mathcal{F} \text{ is fresh} \\
R_{\mathsf{def}} &: \langle V, \Gamma \uplus p(\vec{y})\rangle \rightarrow_{\mathrm{D}} \langle V, \Gamma \uplus A_P[\vec{x}/\vec{y}]\rangle \text{ where } p(\vec{x}) :: A_P \in \mathrm{D} \\
R_{\mathsf{then}} &: \frac{\sigma(\Gamma) \vdash c}{\langle V, \Gamma \uplus \mathsf{now}\ c\ \mathsf{then}\ A\rangle \rightarrow_{\mathrm{D}} \langle V, \Gamma \uplus A\rangle} \\
R_{\mathsf{else}} &: \frac{\sigma(\Gamma) \vdash c}{\langle V, \Gamma \uplus \mathsf{now}\ c\ \mathsf{else}\ A\rangle \rightarrow_{\mathrm{D}} \langle V, \Gamma\rangle}
\end{aligned}
$$

$$
R_{\rightsquigarrow_{\mathrm{D}}} :
$$

$$
\frac{\langle V, \Delta \uplus \{\{\mathsf{next}\ B_j \mid j \leq m\}\} \uplus \{\{\mathsf{now}\ c_i\ \mathsf{else}\ A_i \mid c_i \leq n\}\}\rangle \not\rightarrow_{\mathrm{D}}}{\langle V, \Delta \uplus \{\{\mathsf{now}\ c_i\ \mathsf{else}\ A_i \mid c_i \leq n\}\} \uplus \{\{\mathsf{next}\ B_j \mid j \leq m\}\}\rangle \rightsquigarrow_{\mathrm{D}} \langle V, \{\{A_i \mid i \leq n\}\} \uplus \{\{B_j \mid j \leq m\}\}\rangle}
$$

Figure 3.2: Operational semantics of the tcc language

that $\vec{y} \cap bv(A_P) = \emptyset$ and $\vec{x} \cap bv(A_P) = \emptyset$, and in $R_{\widehat{\ }}$, where it ensures that $x \notin bv(A)$, so that the assumption regarding the use of substitutions with agents is respected.

Now we present a third transition relation: the *input-output* relation $\Longrightarrow_{\mathrm{D}}$:

$$
\frac{\langle V, \Gamma \uplus \exists_V c\rangle \rightarrow^*_{\mathrm{D}} \langle W, \Delta\rangle \rightsquigarrow_{\mathrm{D}} \langle W, \Delta'\rangle}{\langle V, \Gamma\rangle \overset{(c, c \sqcup \exists_W \sigma(\Delta))}{\Longrightarrow_{\mathrm{D}}} \langle W, \Delta'\rangle} \tag{3.2.1}
$$

Intuitively, $\langle V, \Gamma\rangle \overset{(c,c')}{\Longrightarrow_{\mathrm{D}}} \langle W, \Delta\rangle$ is interpreted as follows: starting from a configuration $\langle V, \Gamma\rangle$ and an input constraint c, it is produced an output constraint c' and the system moves to the state $\langle W, \Delta\rangle$ in the next time instant. The reader can see how it is necessary a non-banal mechanism to handle the existential quantification of the underlying constraint system.

The transition system $\rightarrow_{\mathrm{D}}$ is not deterministic neither confluent, due to the rule for $x\widehat{\ }$ which introduces new fresh variables. As an immediate consequence, the same holds for $\Longrightarrow_{\mathrm{D}}$. However, if we consider configurations modulo renaming, we obtain confluent transition systems (and deterministic, in the case of $\Longrightarrow_{\mathrm{D}}$). The key property that makes true this assertion is that transitions commute with renaming according to the results presented below.

First of all, Lemma 3.2.1 shows that the transition relation $\rightarrow_{\mathrm{D}}$ commutes with renaming. This makes such transition relation deterministic.

Lemma 3.2.1 *If it is true that* $\langle V, \Gamma \rangle \longrightarrow^* \langle V', \Delta \rangle$ *and* f *is a renaming of free variables, then* $\langle f(V), f(\Gamma) \rangle \longrightarrow^* \langle f(V'), f(\Delta) \rangle$

Proof. The proof is done by case over the principal agent. Here we just show the proof for the $\mathsf{now}\ c\ \mathsf{then}\ A$ and $x\hat{}A$ agents which are the most interesting cases. The rest of the cases can be proved in a similar way.

For the Positive Ask construct, we assume $\langle V, \Gamma \uplus \mathsf{now}\ c\ \mathsf{then}\ A \rangle \longrightarrow \langle V, \Gamma \uplus A \rangle$ with $\sigma(\Gamma) \vdash c$. Then $\sigma(\Gamma)[f] \vdash c[f]$ by the second property of Proposition 1.4.11. Note that $\sigma(\Gamma)[f] = \sigma(\Gamma[f])$ thus $\langle f(V), \Gamma[f] \uplus \mathsf{now}\ c[f]\ \mathsf{then}\ A[f] \rangle \longrightarrow \langle f(V), \Gamma[f] \uplus A[f] \rangle$. Moreover, by inductive hypothesis there is a derivation $\langle f(V), \Gamma[f] \uplus A[f] \rangle \longrightarrow \langle f(V'), \Delta[f] \rangle$. This proves the case for the Positive Ask agent.

For the second considered case we assume $\langle V, \Gamma \uplus x\hat{}A \rangle \longrightarrow \langle V \cup y, \Gamma \uplus A[x/y] \rangle \longrightarrow^* \langle V', \Delta \rangle$. Note that $\langle f(V), \Gamma[f] \uplus x\hat{}\ A[f] \rangle \longrightarrow \langle f(V) \cup \{f(y)\}, \Gamma[f] \uplus A[f][x/f(y)] \rangle$. We know that $f(y) \notin \mathit{fv}(A[f])$ and $x \notin \mathit{fv}(A)$, therefore $A[f][x/f(y)] = A[f \uplus x/y] = A[x/y][f]$. Moreover, by inductive hypothesis we have a derivation $\langle f(V) \cup \{f(y)\}, \Gamma[f] \uplus A[x/y][f] \rangle \longrightarrow^* \langle f(V'), \Delta[f] \rangle$. This proves the second considered case. ■

The transition relation $\rightsquigarrow_D$ can be seen as a function (in [NPV02b] semantics it is introduced in that way) where timing constructs produce a different multiset of agents. We can see that the set of variables V is not modified when we apply the rule for the $\rightsquigarrow_D$ relation. This makes possible to show the second result:

Lemma 3.2.2 *If it is true that* $\langle V, \Gamma \rangle \rightsquigarrow \langle V, \Delta \rangle$ *and* f *is a renaming of free variables, then* $\langle f(V), f(\Gamma) \rangle \rightsquigarrow \langle f(V), f(\Delta) \rangle$.

Proof. This result is a direct consequence of monotonicity of renaming showed in Proposition 1.4.11. ■

Finally, the most important result is Theorem 3.2.3 which involves the input-output transition relation. We show that also this transition relation commutes with renaming.

Theorem 3.2.3 *If* $\langle V, \Gamma \rangle \overset{(c,c')}{\Longrightarrow} \langle V', \Gamma' \rangle$ *and* f *is a renaming of variables in* $\mathcal{F}$*, then* $\langle f(V), f(\Gamma) \rangle \overset{f(c),f(c')}{\Longrightarrow} \langle f(V'), f(\Gamma') \rangle$.

Proof. We assume that $\langle V, \Gamma \rangle \overset{(c,c')}{\Longrightarrow} \langle V', \Gamma' \rangle$. Following the semantics defined in (3.2.1) this assumption means that $\langle V, \Gamma \uplus \exists_V c \rangle \longrightarrow^* \langle V', \Delta \rangle \rightsquigarrow \langle V', \Gamma' \rangle$ and $c' = c \sqcup \exists_{V'} \sigma(\Delta)$. By Lemmas 3.2.1 and 3.2.2 we can say that, if $f' = f_{|V'}$, we have a derivation

$$\langle f'(V), \Gamma[f'] \uplus (\exists_V c)[f'] \rangle \longrightarrow^* \langle f'(V'), \Delta[f'] \rangle \rightsquigarrow \langle f'(V'), \Gamma''[f'] \rangle$$

Note that $(\exists_V c)[f'] = \exists_V c$ and therefore we obtain $\langle W, \Gamma[f'] \rangle \overset{(c,c'')}{\Longrightarrow} \langle W'', \Gamma''[f'] \rangle$ where $c'' = c \sqcup \exists_{W''} \Gamma''[f']$. Note that $\exists_{W''} \Gamma''[f'] = \exists_{V''} \Gamma''$ and therefore $c'' = c'$. By induction we complete the proof. ■

Now, we can consider the equivalence relation $\sim_c$ over configurations such that

$$\langle V, \Gamma\rangle \sim_c \langle U, \Delta\rangle \iff \exists f \text{ renaming s.t. } \\ dom(f) \subseteq U \cup V, f(V) = U \text{ and } f(\Gamma) = \Delta \quad (3.2.2)$$

$\langle V, \Gamma\rangle \sim_c \langle U, \Delta\rangle$ means that the two configurations only differ in the name of the local variables in V and U. Then we can say that $\Longrightarrow_D$, viewed as a transition system over equivalence classes of configurations, is deterministic. Therefore, this justifies the fact we are talking of tcc as a deterministic language.

In the following, we omit the index D from the transition relations when it is not relevant.

3.2.1 Observable Behaviors

Once we have presented the operational semantics for the tcc language, we must to define what we want to observe in a system. We are not interested in the internal machinery of the system given by the transition relations $\longrightarrow$ and $\rightsquigarrow$. The only interesting thing is the interaction of the system with the environment, given by $\Longrightarrow$. From this point of view, the only thing that we can observe is the sequence of constraints given as input or emitted as output.

We call *observation* a finite (possibly empty) sequence of constraints and we denote with **Obs** the set of all observations. Concatenation of observations s and s' is denoted by $s \cdot s'$, while ϵ is the empty sequence. If $s \in \mathbf{Obs}$, $x \in \mathcal{V}$ and f a substitution, we denote with $\exists_x s$ and $s[f]$ the pointwise extension of the corresponding operations on constraints. $l(s)$ denotes the length of the observation s.

An observation represents a sequence of input or output constraints along the time. If $s = c_1, \ldots, c_n$ and $s' = c'_1, \ldots, c'_n$, we write $\langle V, \Gamma\rangle \overset{(s,s')}{\Longrightarrow^*} \langle W, \Delta\rangle$ for

$$\langle V_0, \Gamma_0\rangle \overset{(c_1,c'_1)}{\Longrightarrow} \langle V_1, \Gamma_1\rangle \overset{(c_2,c'_2)}{\Longrightarrow} \cdots \overset{(c_n,c'_n)}{\Longrightarrow} \langle V_n, \Gamma_n\rangle$$

where $V_0 = V$, $\Gamma_0 = \Gamma$, $V_n = W$ and $\Gamma_n = \Delta$.

Let us define an ordering between observations, given by

$$c_1 \ldots c_n \leq_{\mathsf{obs}} d_1 \ldots d_m \iff m \leq n \wedge \forall 1 \leq i \leq m.\ c_i \sqsubseteq d_i \qquad (3.2.3)$$

This is a sort of pointwise extension of entailment between constraints, with the proviso that an aborted process is like a process which always emits *false*, the greatest element of $\mathcal{C}$.

Observations satisfy some basic properties and we show them by the following results:

Lemma 3.2.4 $(\mathbf{Obs}, \leq_{\mathsf{obs}})$ *has a least upper bound for each nonempty set. The greatest element is* ϵ*. Let* β *be the longest common prefix of all the* α*'s. If* $\{\alpha_i\}_{i \in I} \subseteq$ **Obs***, then we can define two cases*

- *for each* $i \in I$*,* $\alpha_i = \beta \cdot c_i \cdot \gamma_i$ *for some* $c_i \in \mathcal{C}$ *and* $\gamma_i \in \mathbf{Obs}$ *the following is true:*
$$\bigvee_i \alpha_i = \beta \cdot (\bigsqcup_i c_i)$$
- *otherwise*
$$\bigvee_i \alpha_i = \beta$$

Proof. The fact that ϵ is the greatest element is trivial. Now we prove that $\bigvee_i \alpha_i$, as we have defined it, is the lowest upper bound. For each $i \in I$, $\alpha_i \leq_{\mathsf{obs}} \beta$ is obvious by the definition, thus β is an upper bound.

Now we must prove that this is the least upper bound. We proceed by contradiction. Suppose that we have an upper bound γ which is greater than $\bigvee_i \alpha_i$. We call $\gamma = e_1 \dots e_m$ and $\beta \cdot (\bigsqcup_i c_i) = d_1 \dots d_n$. Then there are two cases:

- if $\gamma \not\leq_{\mathsf{obs}} \beta \cdot (\bigsqcup_i c_i)$ then $m > n$ or $\exists j \in \{1 \dots m\}$ s.t. $d_j \not\sqsubseteq e_j$. If $m > n$ then γ is not an upper bound. On the other way, if there exists $d_j \not\sqsubseteq e_j$ then it is not satisfied the condition over the common prefix.
- The case for $\gamma \not\leq_{\mathsf{obs}} \beta$ is trivially proved in a similar way.

■

The second property of observables that we can show is presented in the following lemma. It proves the existence of the greatest lower bound for each bounded set of observables.

Lemma 3.2.5 $(\mathbf{Obs}, \leq_{\mathsf{obs}})$ *has a greatest lower bound for each bounded set. The greatest element is* ϵ*. If* $H \subseteq \mathbf{Obs}$ *has a lower bound, then we can define* $\bigwedge H = c_1 \dots c_n$ *where* $n = \max\{l(\alpha) \mid \alpha \in H\}$ *and* $c_i = \bigsqcap H[i]$*.*

Proof. First of all, note that n does exist since H is bounded. It means that there exists β such that $\beta \leq_{\mathsf{obs}} \alpha$ for each $\alpha \in H$. Therefore $l(\beta) \geq l(\alpha)$ for each $\alpha \in H$.

■

The most complete form of observation that we can perform on the system is the relation between input and output constraints. Given a program D.A, we define the *input-output* behavior

$$io(D.A) = \left\{ (s, s') \mid \exists \langle V, \Gamma \rangle .\ \langle \emptyset, A \rangle \overset{(s,s')}{\Longrightarrow}{}^{*} \langle V, \Gamma \rangle \right\} \qquad (3.2.4)$$

However, we can define a weaker form of observation. If we only observe the output constraints, we obtain what generally are called *strongest postconditions*. Formally

$$sp(D.A) = \{s \in \mathbf{Obs} \mid \exists s' \in \mathbf{Obs}.\ (s', s) \in io(D.A)\} \qquad (3.2.5)$$

3.3 Denotational Semantics

In this section we describe the denotational semantics presented in [SJG94a]. In that work, the authors extended the denotational semantics of cc to the new constructs. Observations of a program are defined as the set of resting points or strongest postconditions. A model of tcc is defined as a sequence of models of cc.

If $s, s' \in \mathbf{Obs}$, we write $s \preceq s'$ to denote that s is a prefix of s'. Let P be a subset of $\mathbf{Obs}$ and $s \in P$. We denote with $P \,\mathsf{after}\, s$ the set $\{s' \in \mathbf{Obs} \mid s \cdot s' \in P\}$. In Definition 3.3.1 the notion of tcc *process* is defined.

Definition 3.3.1 ([SJG94b]) *A* process *is a subset* P *of* $\mathbf{Obs}$ *which respects the following conditions:*

- $\epsilon \in P$*,*
- *it is prefixed closed (i.e. if* $s \in P$ *and* $s' \preceq s$*, then* $s' \in P$*);*
- *it is* determinate*, i.e. for each* $s \in P$*,* $P \,\mathsf{after}\, s$ *is determinate.*

We denote with **Proc** the set of all the processes. Letters P or Q range over processes.

In Figure 3.3 the denotational semantics are described by means of semantic equations. $[c]$ stands for the constraint generated by a set of tokens c. Note that in this description of the denotational semantics there is no rule for the Procedure Call. In [SJG94a] the authors do not consider the problem including such agents. In Chapter 4 we introduce the new denotational semantics where an environment is defined in order to enrich the semantics by handling the Procedure Call agent.

$$
\boxed{
\begin{array}{rcl}
[\![\mathsf{skip}]\!] & = & \mathbf{Obs} \\
[\![\mathsf{abort}]\!] & = & \{\epsilon\} \\
[\![\mathsf{c}]\!] & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \supseteq [c]\} \\
[\![A \parallel B]\!] & = & \{s \in \mathbf{Obs} \mid s \in [\![A]\!]_e \wedge [\![B]\!]\} \\
[\![x\widehat{\ }A]\!] & = & \{s \in \mathbf{Obs} \mid \exists X.s = \exists X.t, \text{for some } t \in [\![A]\!]\} \\
[\![\mathsf{now}\; \mathsf{c}\; \mathsf{then}\; A]\!] & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \supseteq [c] \Rightarrow d \cdot s \in [\![A]\!]\} \\
[\![\mathsf{now}\; \mathsf{c}\; \mathsf{else}\; A]\!] & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \not\supseteq [c] \Rightarrow s \in [\![A]\!]\} \\
[\![\mathsf{next}\; A]\!] & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid s \in [\![A]\!]\}
\end{array}
}
$$

Figure 3.3: Denotational Semantics of tcc ([SJG94a])

The skip agent does nothing, thus every sequence of constraints is included in its denotational semantics. The abort agent causes the termination of all concurrent processes, hence the only observation that can be made of it is ϵ. The denotations of the Tell agent contains the empty sequence and the set of sequences where c is the first element. The Parallel agent simply combines two agents, thus the denotational semantics contains all the sequences which belong to both the semantics of the single agents. The semantics of the hiding operator is defined in a similar way as in [SRP91] but in Chapter 4 we will show why this definition is not appropriate. The denotational semantics of the Positive Ask agent contains all sequences satisfying that the constraint c is in the first element of the sequence and the whole sequence is in the denotational semantics of the agent A. The last two agents in Figure 3.3 are the two timing constructs of the language: Negative Ask and Unit Delay. The semantics for the former is quite similar to the Positive Ask case: it contains all sequences whose first element do not contain c and the sequence s (i.e., the sequence not including the first element) is in the denotational semantics of A. The semantics of the latter construct contains all sequences which, discarding its first element, are in the denotations of the agent A.

The problem with the semantics in [SJG94a] is that it is not fully abstract w.r.t. the operational semantics although the opposite is stated in [SJG94a, Theorem 3.1]. We fix this problem in the following chapter by defining a new denotational semantics.

3.4 Applications

The tcc language is suitable for specifying systems where timing is important, for example small reactive systems such as controllers or embedded systems. However, the definition of new constructs derived from the basic ones showed in Figure 3.1 is necessary in order to ease to the user the specification of systems. For example, in [SJG94a, SJG94b] you can find how something similar to the *watchdog* construct of the popular Esterel ([Ber00]) can be defined. A watchdog is a process which executes an agent A while a condition c is *not* entailed. When such condition is satisfied, then the process A is terminated.

There are other interesting constructs such as the *extended wait* behavior, the *suspension-activation* primitives or the *multiform* time. The extended wait agent executes a process A while a condition is entailed by the store, whereas the suspension-activation primitives allow the programmer to control the execution of a process and to suspend or activate it depending on some conditions. Finally, the multiform time makes able to synchronize a process w.r.t. some constraint, i.e., we execute the agent A only when the constraint c is entailed by the store.

Some applications that can be implemented using tcc are a mouse controller, or an automatic teller machine. These systems have been yet specified in the literature by using the tcc language. Furthermore, the tcc language can be extended with defaults that makes possible to maintain defaults values for some variables.

As we have said before, different approaches have been defined in the last years as extensions over time of the cc paradigm. In particular tcc was defined mainly for embedded systems. However, each language was created to specify a different class of systems. For example, the tccp language that we introduce in Chapter 5 is more appropriate to model systems where the non determinism is crucial. Such systems usually are more complex than the embedded systems.

4

New semantics for tcc

There are several advantages in endowing a programming language with a good denotational semantics. First of all, the denotational semantics is typically better suited than the operational one for comparing different languages and/or different programs, since it naturally abstracts from differences in the syntax [Win93]. This is especially true for a language such as tcc whose operational semantics is given by three different transition systems. In comparison, the denotational semantics is obtained by a standard fixpoint computation over a cpo. This also simplifies the goal of proving program properties by static analysis, using standard techniques such as abstract interpretation [CC77]. However, it is important for the denotational semantics to exactly capture the set of observations we are interested in, otherwise it can only provide limited benefits. For example, a static analysis based on the standard denotational semantics for tcc can only be used to prove safety properties, since we start from an over-approximation of the real set of strongest postconditions.

Denotational semantics can also help in the effort of understanding more in detail the logical meaning of tcc programs. The existence of particular relations between the semantic operators (for example, the fact that an operator is the right/left adjoint to some other) can shed some light on the logic which is behind the language.

As we have seen in Chapter 3, the first timed concurrent constraint language was introduced in [SJG94a, SJG94b] with the name of tcc. The problem with tcc is that, although both a denotational and an operational semantics is provided, they do not correspond when the hiding operator is used together with the now c else A construct, which discovers negative information.

This flaw is not discussed in [SJG94b] and actually [SJG94a] incorrectly states the opposite, i.e., that the denotational semantics is fully abstract w.r.t. the operational one. The problem is illustrated in a slightly different language in [NPV02b, Example 5]. In such work, authors also prove that the denotational and operational semantics do coincide if we consider only *locally independent* processes (see [NPV02b] for details). However, not all tcc processes are locally independent.

A locally independent process has the property that the construct now c else A never occurs within the scope of a hiding operator which bounds a free variable in c. However, this condition severely limits the freedom of building modular systems. Actually, it means that two systems (or two modules in the same system) cannot communicate freely through a common private variable: if they try to detect negative information regarding that variable, their denotational and operational semantics differ. For example, the tcc[1] program in [NPV02a] which solves the Post's correspondence problem is not locally independent.

In this chapter we show that the problem is essentially due to the definition of the denotational semantics corresponding to the hiding operator. The semantics provided in [SJG94b, SJG94a, NPV02b] are essentially the same as the semantics in [SRP91] for cc, just extended over time. However, in [SRP91] there is no way of detecting negative information, so that the extended definition only works when now c else A is not used. In this work, without changing the semantic domain, we define a new semantic operator for hiding: so patched, the semantics is fully abstract w.r.t. the operational behavior.

At the end of this chapter we provide an initial study of the expressive power of the now c else A construct by the point of view of the denotational semantics.

4.1 New denotational semantics

The denotational semantics given in [SJG94a] is only an over approximation of the real operational behavior. We will show in this section that the problem lies in the denotational treatment of the hiding operator, and that a different definition gives rise to a denotational semantics which is fully-abstract w.r.t. the observable of strongest postconditions.

We use partially the definitions of Section 3.3. If $\mathsf{s}, \mathsf{s}' \in \mathbf{Obs}$, we write $\mathsf{s} \preceq \mathsf{s}'$ to denote that s is a prefix of s'. Let P be a subset of $\mathbf{Obs}$ and $\mathsf{s} \in \mathsf{P}$. We denote with $\mathsf{P}\,\mathsf{after}\,\mathsf{s}$ the set $\{\mathsf{s}' \in \mathbf{Obs} \mid \mathsf{s} \cdot \mathsf{s}' \in \mathsf{P}\}$. Moreover, we denote with $\mathsf{P}[1] = \{\mathsf{c} \in \mathcal{C} \mid \mathsf{c} \cdot \mathsf{s} \in \mathsf{P}\}$. Then we can redefine what a process is:

Definition 4.1.1 *A* process *is a subset* P *of* **Obs** *which fulfill the following conditions:*

- $\epsilon \in \mathsf{P}$,
- *it is prefixed closed (i.e., if* $\mathsf{s} \in \mathsf{P}$ *and* $\mathsf{s}' \preceq \mathsf{s}$*, then* $\mathsf{s}' \in \mathsf{P}$*);*
- *it is* determinate*, i.e., for each* $\mathsf{s} \in \mathsf{P}$*,* $(\mathsf{P}\,\mathsf{after}\,\mathsf{s})[1]$ *is a partial Moore family.*

We denote with **Proc** the set of all the processes. If P is a process and $\mathsf{s} \in \mathbf{Obs}$, we denote with $\overline{\mathsf{P}[1]}$ the partial closure operator on $\mathcal{C}$ whose set of fixpoints is $\mathsf{P}[1]$. We can view a process P as an operator $\bar{\mathsf{P}}$ over **Obs** by defining

$$\bar{\mathsf{P}}(\mathsf{s}) = \begin{cases} \mathsf{c}' \cdot \overline{(\mathsf{P}\,\mathsf{after}\,\mathsf{c}')}(\mathsf{s}') & \text{if } \mathsf{s} = \mathsf{c} \cdot \mathsf{s}' \text{ and } \mathsf{c}' = \overline{\mathsf{P}[1]}(\mathsf{c}) \text{ is defined,} \\ \epsilon & \text{otherwise.} \end{cases} \tag{4.1.1}$$

[1] Actually, it is a program written in the deterministic fragment of ntcc, which can be immediately translated to tcc.

The idea is that, in deterministic tcc, it is possible to recover the input-output behavior of an agent from its postconditions. If P is the set of postconditions of an agent, then $\bar{\mathsf{P}}$ is the corresponding input-output behavior. This is formalized by the following result:

Theorem 4.1.2 *Given a program* $\mathsf{D.A}$*,* $sp(\mathsf{D.A}) \in \mathbf{Proc}$ *and* $\overline{sp(\mathsf{D.A})} = io(\mathsf{D.A})$*.*

Proof. Here we only present the sketch of the proof. We must show on one hand that $sp(\mathsf{D.A})$ is a process. This is obvious by the definition of $sp(\mathsf{D.A})$ and processes. In addition, we have to show that the closure of $sp(\mathsf{D.A})$ coincides with the input-output behavior. Due to the deterministic nature of processes, the partial closure operator and their definitions, it can be trivially proved. ■

Given a sequence s, we denote with $l(\mathsf{s})$ its length. Given a process P, the *divergences* of P are those sequences s such that $l(\bar{\mathsf{P}}(\mathsf{s})) < l(\mathsf{s})$. The set of divergences of P is denoted by $d(\mathsf{P})$. We write $\bar{\mathsf{P}}(\mathsf{s}) \uparrow$ when s is a divergence for P and $\bar{\mathsf{P}}(\mathsf{s}) \downarrow$ otherwise. Note that if $\mathsf{P} \subseteq \mathsf{Q}$ then $d(\mathsf{Q}) \subseteq d(\mathsf{P})$.

Processes can be ordered by inclusion or by the following *divergence ordering.* We say

$$\mathsf{P} \leq_{\mathsf{div}} \mathsf{Q} \iff \mathsf{P} \subseteq \mathsf{Q} \subseteq \mathsf{P} \cup d(\mathsf{P}) \tag{4.1.2}$$

$(\mathbf{Proc}, \leq_{\mathsf{div}})$ is a cpo. The least element is the process $\{\epsilon\}$, while the join of chains is given by union. A similar ordering has been already used in [SRP91] for cc.

In [SJG94a, NPV02b] inclusion is the order relation used to compute the denotational semantics as the least fixpoint of the semantic operators. However, we will need to use the more refined divergence ordering since the semantic operator that we are going to introduce for the hiding agent, is not monotone w.r.t. inclusion.

Processes satisfy some additional interesting properties. In Proposition 4.1.3 we show that processes are idempotent and extensive over $(\mathbf{Obs}, \leq_{\mathsf{div}})$. Moreover, in Lemma 4.1.4 we prove a weak form of monotonicity for processes. These properties will be very useful for the definition of the new semantics for hiding.

Proposition 4.1.3 *Let* P *be a process, then* $\bar{\mathsf{P}}$ *is idempotent and extensive over* $(\mathbf{Obs}, \leq_{\mathsf{obs}})$*.*

Proof. We proceed by induction on the length of an observation α and we prove that $\bar{\mathsf{P}}(\alpha) \geq_{\mathsf{obs}} \alpha$.

If $\alpha = \epsilon$, by definition $\bar{\mathsf{P}}(\alpha) = \epsilon \geq_{\mathsf{obs}} \epsilon$. Otherwise, if $\alpha = \mathsf{c} \cdot \alpha'$ then we have two cases:

- $\bar{\mathsf{P}}(\alpha) = \mathsf{c}' \cdot \overline{\mathsf{P\,after\,c}'}(\alpha')$ where $\mathsf{c}' = \overline{\mathsf{P}[1]}(\mathsf{c})$. Since $\overline{\mathsf{P}[1]}$ is a partial closure operator, then $\mathsf{c}' \sqsupseteq \mathsf{c}$. By inductive hypothesis, $\overline{\mathsf{P\,after\,c}'}(\alpha') \geq_{\mathsf{obs}} \alpha'$. It is immediate to check that $\bar{\mathsf{P}}(\alpha) \geq_{\mathsf{obs}} \mathsf{c} \cdot \alpha' = \alpha$.

- $\overline{\mathsf{P}[1]}(\mathsf{c})$ is not defined. Then $\bar{\mathsf{P}}(\alpha) = \epsilon \geq_{\mathsf{obs}} \alpha$.

We now prove by induction on the length of the observations, that $\bar{P}(\bar{P}(\alpha)) = \bar{P}(\alpha)$. The base case for $\alpha = \epsilon$ is trivial. Otherwise, $\alpha = c \cdot \alpha'$ and we have two cases:

- $\bar{P}(\alpha) = c' \cdot \overline{\mathsf{P\,after\,c'}}(\alpha')$ where $c' = \overline{P[1]}(c)$. Since $\overline{P[1]}$ is a partial closure operator, then it is idempotent on c and $c' = \overline{P[1]}(c')$. Therefore $\bar{P}(\bar{P}(\alpha)) = c' \cdot \overline{\mathsf{P\,after\,c'}}(\overline{\mathsf{P\,after\,c'}}(\alpha'))$. By inductive hypothesis we have $\bar{P}(\bar{P}(\alpha)) = c' \cdot \overline{\mathsf{P\,after\,c'}}(\alpha') = \bar{P}(\alpha)$.
- if $\overline{P[1]}(c)$ is not defined then $\bar{P}(\alpha) = \epsilon$ and $\bar{P}(\bar{P}(\alpha)) = \epsilon = \bar{P}(\alpha)$.

■

Note that $\bar{P}$ is not monotone w.r.t. $\leq_{\mathsf{obs}}$. For example consider $P = \{\epsilon, c, c \cdot \mathit{true}, \mathit{true}\}$ with $c \sqsupset \mathit{true}$. If $\alpha = c \cdot \mathit{true}$ and $\beta = \mathit{true} \cdot \mathit{true}$, we have $\bar{P}(\alpha) = c \cdot \mathit{true}$ and $\bar{P}(\beta) = \mathit{true}$. Note that $\beta <_{\mathsf{obs}} \alpha$ but $\bar{P}(\beta) >_{\mathsf{obs}} \bar{P}(\alpha)$. However, $\bar{P}$ enjoys the weak form of monotonicity shown in the following lemma.

Lemma 4.1.4 *If* $\alpha \leq_{\mathsf{obs}} \beta \leq_{\mathsf{obs}} \bar{P}(\alpha)$, *then* $\bar{P}(\beta) = \bar{P}(\alpha)$.

Proof. We proceed by induction on the length of α. If $\alpha = \epsilon$ and $\epsilon \leq_{\mathsf{obs}} \beta$, then $\beta = \epsilon$ and the property trivially holds. Otherwise, $\alpha = c \cdot \alpha'$ and there are two cases:

- $\bar{P}(\alpha) = c' \cdot \overline{\mathsf{P\,after\,c'}}(\alpha')$ where $c' = \overline{P[1]}(c)$. Since $\alpha \leq_{\mathsf{obs}} \beta \leq_{\mathsf{obs}} \bar{P}(\alpha)$, then $\beta = c'' \cdot \beta'$ which $c \sqsubseteq c'' \sqsubseteq c'$. Since $\overline{P[1]}$ is a partial closure operator, then $\overline{P[1]}(c'') = c$ and therefore $\bar{P}(\beta) = c' \cdot \overline{\mathsf{P\,after\,c'}}(\beta')$. Moreover, $\alpha \leq_{\mathsf{obs}} \beta' \leq_{\mathsf{obs}} \bar{P}(\alpha)$, and the thesis follows by inductive hypothesis.
- if $\overline{P[1]}(c)$ is not defined, then $\bar{P}(\alpha) = \epsilon$. Therefore, either $\beta = \epsilon$ or $\beta = c' \cdot \beta'$ with $c' \sqsupseteq c$. In the first case, it trivially holds. $\bar{P}(\beta) = \epsilon$. In the latter, since $\overline{P[1]}$ is a partial closure operator, then $\overline{P[1]}(c')$ is undefined, and $\bar{P}(\beta) = \epsilon$ too.

■

Moreover, we can say that it is not true that if $P \subseteq Q$ then $P(\alpha) \geq_{\mathsf{obs}} Q(\alpha)$ for each $\alpha \in$ **Obs**. Assume given $c_0, c_1 \in \mathcal{C}$ with $c_0 \sqsubset c_1$ and $P = \{\epsilon, c_1, c_1 \cdot c_1\}$, $Q = P \cup \{c_0\}$. Then for $\alpha = c_0 \cdot c_1$ we have $\bar{P}(\alpha) = c_1 \cdot c_1$ and $\bar{Q}(\alpha) = c_0$ and it is not true that $c_1 \cdot c_1 \geq_{\mathsf{obs}} c_0$.

In order to handle recursion, we define the concept of *environment* as a mapping from process identifiers to processes. We denote with **Env** the set of all the environments. We also consider an infinite sequence $\vec{\alpha} = \alpha_1 \ldots \alpha_n \ldots$ of variables. Each process identifier p of arity n is mapped by the environment in a process on the free variables $\alpha_1, \ldots, \alpha_n$, so that the formal parameters of the definition of p are not relevant. When we write a substitution $\{\vec{x}/\vec{\alpha}\}$ or $\{\vec{\alpha}/\vec{x}\}$, we are actually denoting with $\vec{\alpha}$ not the entire infinite sequence, but the initial prefix which has the same length as $\vec{x}$.

Therefore, the denotational semantics of an agent A is a map

$$\llbracket \mathsf{A} \rrbracket : \mathbf{Env} \to \mathbf{Proc}$$

defined by induction on the structure of the agents in Figure 4.1

$$\begin{array}{rcl}
\llbracket \mathsf{c} \rrbracket_e & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \sqsupseteq c\} \\
\llbracket \mathsf{skip} \rrbracket_e & = & \mathbf{Obs} \\
\llbracket \mathsf{abort} \rrbracket_e & = & \{\epsilon\} \\
\llbracket \mathsf{A} \parallel \mathsf{B} \rrbracket_e & = & \{s \in \mathbf{Obs} \mid s \in \llbracket A \rrbracket_e \cap \llbracket B \rrbracket_e\} \\
\llbracket x^\frown A \rrbracket_e & = & \exists_x \llbracket A \rrbracket_e \\
\llbracket p(\vec{x}) \rrbracket_e & = & e(p)[\vec{\alpha}/\vec{x}] \\
\llbracket \mathsf{now}\ c\ \mathsf{then}\ A \rrbracket_e & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \sqsupseteq c \Rightarrow d \cdot s \in \llbracket A \rrbracket_e\} \\
\llbracket \mathsf{now}\ c\ \mathsf{else}\ A \rrbracket_e & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \not\sqsupseteq c \Rightarrow s \in \llbracket A \rrbracket_e\} \\
\llbracket \mathsf{next}\ A \rrbracket_e & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid s \in \llbracket A \rrbracket_e\}
\end{array}$$

Figure 4.1: New Denotational Semantics of tcc

The cases are similar to those defined in Figure 3.3. For further details on the way the denotational semantic works you can see [SJG94a]. Since we explicitly deal with recursion, we need to add an environment to all the semantic functions. Thus, there are two main differences between both the old and the new semantics, one is the fact that we handle recursion, and the other is that we give a different denotational semantics for the Hiding construct.

Next we explicitly analyze the case of the existential quantifier, which differs from the ways it is treated in the literature. We have defined

$$\llbracket x^\frown A \rrbracket_e = \exists_x \llbracket A \rrbracket_e$$

where we define the existential quantification over processes as

$$\exists_x P = \{s \mid \exists_x.s = \exists_x.\bar{P}(s'), s' \in \mathbf{Obs}, \exists x.s' = s'\} \tag{4.1.3}$$

The intuitive idea is that s is an output sequence for $\exists_x P$ iff there exists an input sequence s', which does not contain any information about x (i.e., $\exists_x s' = s'$), such that the output of P on s' is equal to s, modulo constraints over x. The key differences w.r.t. the treatment of existential quantifiers in [SJG94a, NPV02b] is that we consider P as an input-output operator, so that we can isolate the particular output sequences which come as the results from input sequences without information on x.

The operator $\exists_x$ defined in (4.1.3) can be presented in an alternative form, which is probably less intuitive but easier to handle. This is given by the following result:

Proposition 4.1.5 *If* P *is a process then*

$$\exists_x P = \{s \in \mathbf{Obs} \mid \exists_x.s = \exists_x.\bar{P}(\exists_x.s)\}$$

Proof. If $s \in \exists_x P$ there exists $s' \in \mathbf{Obs}$ such that $\exists_x s' = s'$ and $\exists_x s = \exists_x \bar{P}(s')$. Then, since we know by Proposition 4.1.3 that $\bar{P}$ is extensive, $\exists_x.s \geq_{\mathsf{obs}} \exists_x.s' = s'$ and moreover $\exists_x s \leq_{\mathsf{obs}} \bar{P}(s')$. Therefore, by Lemma 4.1.4, $\exists_x.\bar{P}(\exists_x.s) = \exists_x.\bar{P}(s') = \exists_x s$.

The converse implication is trivial. If $\exists_x s = \exists_x \bar{P}(\exists_x s)$ it is enough to take $s' = \exists_x s$ to obtain $s \in \exists_x P$ according to the definition. ∎

Using this characterization we can easily prove that $[\![x\hat{\ }A]\!]_e$ is a process when $[\![A]\!]_e$ is a process.

Proposition 4.1.6 $\exists_x P$ *is a process when* P *is a process.*

Proof. First of all, we know that $\epsilon \in \exists_x P$ since $\bar{P}(\exists_x \epsilon) = \bar{P}(\epsilon) = \epsilon$. Then, if $s \in \exists_x P$ we can say that $\exists_x s = \exists_x \bar{P}(\exists_x s)$. If we take $s = t_1 \cdot t_2$ with t_1 of length n, then it is true that $\exists_x t_1 = \exists_x \bar{P}(\exists_x t_1)$ and therefore $t_1 \in \exists_x P$.

Finally, if $s \in \exists_x P$, we need to prove $(\exists_x P\ \mathsf{after}\ s)[1]$ is a partial Moore family. Therefore consider $S \subseteq \exists_x P\ \mathsf{after}\ s$ and assume S is not empty. If $c \in S$ then $s \cdot c \in \exists_x P$ and therefore $\exists_x \bar{P}(\exists_x s \cdot \exists_x c) = \exists_x s \cdot \exists_x c$. If we denote with $\rho = \overline{P\ \mathsf{after}\ \bar{P}(\exists_x s)}$, this means that $\exists_x \rho(\exists_x c) = \exists_x c$. We only need to prove that given a family $\{c_i\}_{i \in I} \subseteq S$ we have $\exists_x \rho(\exists_x(\bigsqcap c_i)) = \exists_x \bigsqcap c_i$ but this is a direct consequence of the meet-additivity of $\exists_x$ and upper closure operators. Thus, the theorem is proved. ∎

We can define some additional properties that are satisfied by processes:

Lemma 4.1.7 *The following properties are satisfied:*

1. *If* $P \subseteq Q$ *then* $[\![\mathsf{now}\ c\ \mathsf{else}\ P]\!]_e \subseteq [\![\mathsf{now}\ c\ \mathsf{else}\ Q]\!]_e$.

2. *If* $P \subseteq Q$ *then* $[\![\mathsf{now}\ c\ \mathsf{then}\ P]\!]_e \subseteq [\![\mathsf{now}\ c\ \mathsf{then}\ Q]\!]_e$.

3. *If* $P \subseteq Q$ *then* $[\![\mathsf{next}\ P]\!]_e \subseteq [\![\mathsf{next}\ Q]\!]_e$

4. *If* $P_1 \subseteq Q_1$ *and* $P_2 \subseteq Q_2$ *then* $P_1 \cap P_2 \subseteq Q_1 \cap Q_2$.

where we are abusing notation by using processes inside the semantic brackets instead of agents.

Proof. Trivially follows from the definitions. ∎

Note that it is not true in general that $\mathsf{P} \subseteq \mathsf{Q}$ implies $\exists_x \mathsf{P} \subseteq \exists_x \mathsf{Q}$. For example, consider $\mathsf{P} = \{\epsilon, x = 0, x = 0 \cdot \mathit{true}\}$ and $\mathsf{Q} = \mathsf{P} \cup \{x = 0 \vee x = 1\}$. Then $\mathit{true} \cdot \mathit{true} \in \exists_x \mathsf{P}$ since $\exists_x \bar{\mathsf{P}}(\mathit{true} \cdot \mathit{true}) = \exists_x(x = 0 \cdot \mathit{true}) = \mathit{true} \cdot \mathit{true}$. However $\exists_x \bar{\mathsf{Q}}(\mathit{true} \cdot \mathit{true}) = \exists_x(x = 0 \vee x = 1) = \mathit{true} \neq \mathit{true} \cdot \mathit{true}$. This is the reason why we have introduced the $\leq_{\mathsf{div}}$ ordering.

To define the semantics of a program, we first extend the ordering given by $\leq_{\mathsf{div}}$ over processes pointwise to environments. Then, given a program $D.A$, we define a map from environments to environments given by

$$[\![\mathsf{D}]\!]_e = \lambda p. \begin{cases} [\![A_p]\!]_e[\vec{x}/\vec{\alpha}] & \text{if } p(\vec{x}) :: A_p \in D \\ e(p) & \text{otherwise} \end{cases}$$

Then, the semantics of a program $D.A$ is defined as

$$[\![\mathsf{D}.A]\!] = [\![A]\!]_{\mathit{lfp}[\![\mathsf{D}]\!]}$$

Since here we use fixed points, to make sense of this definition, we need to prove that all the semantic operators for agents are monotonic. In order to reach our purpose, we first prove some results that are used in such proof. In Lemma 4.1.8 we show that if $\mathsf{P} \leq_{\mathsf{div}} \mathsf{Q}$, then for each sequence of observables, the closure of both processes coincides or it is a divergence.

Lemma 4.1.8 *If* $\mathsf{P} \leq_{\mathsf{div}} \mathsf{Q}$ *then, for each* $s \in$ **Obs**, *either* $\bar{\mathsf{P}}(s) = \bar{\mathsf{Q}}(s)$ *or* $\bar{\mathsf{P}}(s) \uparrow$.

Proof. The proof proceeds by induction on the length of s. If $s = \epsilon$ the property is trivial. Otherwise $s = c \cdot s'$ and there are two cases:

- $\bar{\mathsf{P}}(s) = c' \cdot \overline{\mathsf{P}\,\mathsf{after}\,c'}(s')$ where $c' = \overline{\mathsf{P}[1]}(c)$. Since $\mathsf{P} \subseteq \mathsf{Q} \subseteq d(\mathsf{P})$, and c' is not a divergence for $\mathsf{P}[1]$, then $\mathsf{Q}[1] \subseteq \mathsf{P}[1] \cup \{c'' \mid c'' \sqsupset c'\}$. It follows that $c' = \overline{\mathsf{Q}[1]}(c)$ and therefore $\bar{\mathsf{Q}}(s) = c' \cdot \overline{\mathsf{Q}\,\mathsf{after}\,c'}(s')$. It is enough to prove that $\mathsf{P}\,\mathsf{after}\,c' \leq_{\mathsf{div}} \mathsf{Q}\,\mathsf{after}\,c'$. It trivially holds $\mathsf{P}\,\mathsf{after}\,c' \subseteq \mathsf{Q}\,\mathsf{after}\,c'$ since $\mathsf{P} \subseteq \mathsf{Q}$. Moreover, if $t \in (\mathsf{Q} \setminus \mathsf{P})\,\mathsf{after}\,c'$ then $c' \cdot t \in \mathsf{Q} \setminus \mathsf{P} \subseteq d(\mathsf{P})$ and therefore $t \in d(\mathsf{P}\,\mathsf{after}\,c')$ since $\bar{\mathsf{P}}(c') \downarrow$. The thesis follows by induction.
- $\overline{\mathsf{P}[1]}(c)$ is not defined, hence $\bar{\mathsf{P}}(s) \uparrow$.

■

The next step is to prove that processes form a cpo. It is proved in Lemma 4.1.10. However, we first prove the following result which will be used in the proof of such lemma.

Lemma 4.1.9 *If* $\{\mathsf{P}_i\}_{i \leq \alpha}$ *is a chain*[2] *of processes, then* $d(\bigcup_i \mathsf{P}_i) = \bigcap_i d(\mathsf{P}_i)$.

[2] See Chapter 1 for the definition of chain.

Proof. Let $\mathsf{P} = \bigcup_i \mathsf{P}_i$. Since $\mathsf{P} \geq_{\mathsf{div}} \mathsf{P}_i$ it is obvious that $d(\mathsf{P}) \subseteq \bigcap_i d(\mathsf{P}_i)$. We want to prove the other inequality, i.e.,

$$d(\mathsf{P}) \supseteq \bigcap_i d(\mathsf{P}_i)$$

We actually prove the opposite, i.e., that $s \notin d(\mathsf{P})$ implies there exists i such that $s \notin d(\mathsf{P}_i)$. The proof is by induction on the length of s.

For $s = \epsilon$ the property is trivially true. Otherwise $s = s' \cdot c$. Since $s \notin d(\mathsf{P})$ then $s' \notin d(\mathsf{P})$ and therefore there exists i such that $s' \notin d(\mathsf{P}_i)$. Since $\mathsf{P} \geq_{\mathsf{div}} \mathsf{P}_i$, it is also the case that $\bar{\mathsf{P}}(s') = \bar{\mathsf{P}}_i(s')$. Moreover, $\bar{\mathsf{P}}(s) = \bar{\mathsf{P}}(s') \cdot d$ where $d = \overline{\mathsf{P}\,\mathsf{after}\,\bar{\mathsf{P}}(s')}(c) \sqsupseteq c$. This implies that there exists j such that $\bar{\mathsf{P}}(s) \in \mathsf{P}_j$. Given $h \geq \max(i, j)$, we have

$$\bar{\mathsf{P}}_h(s) = \bar{\mathsf{P}}(s') \cdot \overline{\mathsf{P}_h\,\mathsf{after}\,\bar{\mathsf{P}}(s')}(c)$$

Since $\bar{\mathsf{P}}(s) \in \mathsf{P}_h$, then $d \in (\mathsf{P}_h\,\mathsf{after}\,\bar{\mathsf{P}}(s'))[1]$ and therefore $\overline{\mathsf{P}_h\,\mathsf{after}\,\bar{\mathsf{P}}(s')}(c) \downarrow$. This completes the proof. ■

Lemma 4.1.10 (**Proc**, $\leq_{\mathsf{div}}$) *is a cpo.*

Proof. We first prove that $\leq_{\mathsf{div}}$ is a partial order. The reflexive and anti-symmetric properties are obvious. Now assume $\mathsf{P} \leq_{\mathsf{div}} \mathsf{Q}$ and $\mathsf{Q} \leq_{\mathsf{div}} \mathsf{R}$. Note that $d(\mathsf{Q}) \subseteq d(\mathsf{P})$ since if $\bar{\mathsf{P}}(s)\downarrow$ then $\bar{\mathsf{Q}}(s)\downarrow$ by Lemma 4.1.8. Therefore $\mathsf{P} \subseteq \mathsf{R}$ and $\mathsf{R} \subseteq \mathsf{Q} \cup d(\mathsf{Q}) \subseteq \mathsf{P} \cup d(\mathsf{P}) \cup d(\mathsf{Q}) = \mathsf{P} \cup d(\mathsf{P})$, whence $\mathsf{P} \leq_{\mathsf{div}} \mathsf{R}$.

Note that $d(\{\epsilon\}) = \mathbf{Obs} \setminus \{\epsilon\}$. Therefore, for each process P, $\{\epsilon\} \leq_{\mathsf{div}} \mathsf{P}$. Given a chain $\{\mathsf{P}_i\}_{i \leq \alpha}$ of processes, we want to prove that $\mathsf{P} = \bigcup_i \mathsf{P}_i$ is their lowest upper bound.

First we prove that it is a process. Closure by prefix and containment of ϵ are obvious. We only need to prove that if $s \in \mathsf{P}$, then $\mathsf{P}\,\mathsf{after}\,s$ is determinate. First of all, note that if $h \geq k$, $c \in \mathsf{P}_h\,\mathsf{after}\,s$ but $c \notin \mathsf{P}_k\,\mathsf{after}\,s$, it means that $s \cdot c \in d(\mathsf{P}_k)$. As a result, if $s \cdot d \in \mathsf{P}_k$, then $c \not\sqsupseteq d$.

Now, consider $\mathsf{S} \subseteq \mathsf{P}\,\mathsf{after}\,s$ and let $\mathsf{S}_i = \mathsf{S} \cap \mathsf{P}_i\,\mathsf{after}\,s$ for each $i \leq \alpha$. Let k be the first index such that $\mathsf{S}_k \neq \emptyset$ and $h \geq k$. By the condition of determinacy and the observation before either $\bigsqcap \mathsf{S}_k \in \mathsf{P}_h\,\mathsf{after}\,s$ or $\bigsqcap \mathsf{S}_k \not\leq_{\mathsf{div}} \bigsqcap \mathsf{S}_h$. However, the second condition is not possible since $\mathsf{S}_k \supseteq \mathsf{S}_h$, therefore $\bigsqcap \mathsf{S}_k \in \mathsf{P}_h\,\mathsf{after}\,s$ for each $k \geq h$. Since $\bigsqcap \mathsf{S} = \bigsqcap_{h \leq i \leq \alpha} \bigsqcap \mathsf{S}_i$, we have $\bigsqcap \mathsf{S} \in \mathsf{P}_h\,\mathsf{after}\,s \subseteq \mathsf{P}\,\mathsf{after}\,s$.

Now, to prove that we really have a lowest upper bound, we begin to check that $\mathsf{P}_i \leq_{\mathsf{div}} \mathsf{P}$ for each $i \leq \alpha$. It is obvious that $\mathsf{P}_i \subseteq \mathsf{P}$. Moreover, if $s \in \mathsf{P} \setminus \mathsf{P}_i$ then $s \in \mathsf{P}_j$ for some $j \geq i$ and therefore $s \in d(\mathsf{P}_i)$. Therefore P is an un upper bound. We need to prove it is the least. If R is a process such that $\mathsf{P}_i \leq_{\mathsf{div}} \mathsf{R}$ for each $i \leq \alpha$, we have $\mathsf{R} \supseteq \bigcup_i \mathsf{P}_i$. If $s \in \mathsf{R} \setminus \bigcup_i \mathsf{P}_i$, then $s \in d(\mathsf{P}_i)$ and, by Lemma 4.1.9, this means $s \in d(\mathsf{P})$, which is all that we need to state that $\mathsf{P} \leq_{\mathsf{div}} \mathsf{R}$. ■

We can finally prove that the existential quantification is monotone and continuous. These properties are presented in Theorem 4.1.11 and Theorem 4.1.12. These results allow us to define the denotational semantics of programs in terms of fixed points.

Theorem 4.1.11 (Existential Quantification is Monotonic) *If* $\mathsf{P} \leq_{\mathsf{div}} \mathsf{Q}$ *then* $\exists_{\mathsf{x}}\mathsf{P} \leq_{\mathsf{div}} \exists_{\mathsf{x}}\mathsf{Q}$.

Proof. Note that if $\mathsf{P} \subseteq \mathsf{Q}$, then $\mathsf{P}(s) \geq_{\mathsf{div}} \mathsf{Q}(s)$ for each $s \in \mathsf{Obs}$. Therefore, if we assume that $s \in \exists_{\mathsf{x}}\mathsf{P}$, then we have $\exists_{\mathsf{x}}.\mathsf{Q}(\exists_{\mathsf{x}}s) \leq_{\mathsf{div}} \exists_{\mathsf{x}}.\mathsf{P}(\exists_{\mathsf{x}}s) = \exists_{\mathsf{x}}s$. The opposite inequality is obvious from the extensibility of Q. We now prove that $\exists_{\mathsf{x}}\mathsf{Q} \subseteq \exists_{\mathsf{x}}\mathsf{P} \cup d(\exists_{\mathsf{x}}\mathsf{P})$. Assume $s \in \exists_{\mathsf{x}}\mathsf{Q} \setminus \exists_{\mathsf{x}}\mathsf{P}$. Then $\exists_{\mathsf{x}}s = \exists_{\mathsf{x}}\mathsf{Q}(\exists_{\mathsf{x}}s) <_{\mathsf{div}} \exists_{\mathsf{x}}\mathsf{P}(\exists_{\mathsf{x}}s)$. Since $\mathsf{P}(\exists_{\mathsf{x}}s) \neq \mathsf{Q}(\exists_{\mathsf{x}}s)$, then $\mathsf{P}(\exists_{\mathsf{x}}s)$ has a length strictly less than n, the length of $\exists_{\mathsf{x}}s$. Therefore, for each $s' \geq_{\mathsf{obs}} s$, $\mathsf{P}(\exists_{\mathsf{x}}s')$ has a length strictly less than n. Since

$$(\exists_{\mathsf{x}}\mathsf{P})(s) = \bigsqcap\{s' \geq_{\mathsf{obs}} s \mid \exists_{\mathsf{x}}s' = \exists_{\mathsf{x}}\mathsf{P}(\exists_{\mathsf{x}}s')\}$$

each of the s' we consider for computing the meet has a length less then n, and therefore the same holds for $(\exists_{\mathsf{x}}\mathsf{P})(s)$ too. This proves that s is a divergence for $\exists_{\mathsf{x}}.\mathsf{P}$. ■

Finally, we can say that the existential quantification is continuous.

Theorem 4.1.12 (Existential quantification is Continuous) $\exists_{\mathsf{x}}$ *is continuous in* $(\mathbf{Proc}, \leq_{\mathsf{div}})$.

Proof. We already proved it is monotone. To show continuity, assume $\{\mathsf{P}_i\}_{i \leq \alpha}$ is a chain in **Proc**. We want to prove that

$$\exists_{\mathsf{x}} \bigcup_i \mathsf{P}_i \subseteq \bigcup_i \exists_{\mathsf{x}}\mathsf{P}_i$$

Let $\mathsf{P} = \bigcup_i \mathsf{P}_i$ and assume $s \in \exists_{\mathsf{x}}\mathsf{P}$. It means that $\exists_{\mathsf{x}}s = \exists_{\mathsf{x}}\bar{\mathsf{P}}(\exists_{\mathsf{x}}s)$. We want to prove that there exists an $i \leq \alpha$ such that $\exists_{\mathsf{x}}s = \exists_{\mathsf{x}}\bar{\mathsf{P}}_i(\exists_{\mathsf{x}}s)$. By Lemma 4.1.8, it is enough to prove that there exists $i \leq \alpha$ such that $\bar{\mathsf{P}}_i(\exists_{\mathsf{x}}s) \downarrow$. But this immediately follows from Lemma 4.1.9. ■

Moreover, we can prove the following properties for the new semantics which, together with the previous result allow us to prove that the semantics of processes are continuous.

Lemma 4.1.13 *The following properties are satisfied:*

1. *If* $\mathsf{P} \leq_{\mathsf{div}} \mathsf{Q}$ *then* $[\![\mathsf{now}\ c\ \mathsf{else}\ \mathsf{P}]\!]_e \leq_{\mathsf{div}} [\![\mathsf{now}\ c\ \mathsf{else}\ \mathsf{Q}]\!]_e$.
2. *If* $\mathsf{P} \leq_{\mathsf{div}} \mathsf{Q}$ *then* $[\![\mathsf{now}\ c\ \mathsf{then}\ \mathsf{P}]\!]_e \leq_{\mathsf{div}} [\![\mathsf{now}\ c\ \mathsf{then}\ \mathsf{Q}]\!]_e$.
3. *If* $\mathsf{P} \leq_{\mathsf{div}} \mathsf{Q}$ *then* $[\![\mathsf{next}\ \mathsf{P}]\!]_e \leq_{\mathsf{div}} [\![\mathsf{next}\ \mathsf{Q}]\!]_e$

4. If $\mathsf{P}_1 \leq_{\mathsf{div}} \mathsf{Q}_1$ *and* $\mathsf{P}_2 \leq_{\mathsf{div}} \mathsf{Q}_2$ *then* $\mathsf{P}_1 \cap \mathsf{P}_2 \leq_{\mathsf{div}} \mathsf{Q}_1 \cap \mathsf{Q}_2$.

where we are abusing notation by using processes inside the semantic brackets instead of agents.

Proof. We prove each sentence separately:

Point 1 The monotonicity w.r.t. $\subseteq$ is obvious. We proceed by contradiction, thus assume that $s = d \cdot s' \in [\![\mathsf{now\ c\ else\ Q}\,]\!]_e \setminus [\![\mathsf{now\ c\ else\ P}]\!]_e$. If $d \not\vdash c$ then $s' \in Q \setminus P$, and therefore $P(s') \uparrow$. As a result, for each $s'' \geq_{\mathsf{obs}} s$, if $s'' \in [\![\mathsf{now\ c\ else\ P}]\!]_e$ then or $s'' \geq_{\mathsf{obs}} P(s)$ and therefore $l(s'') \leq l(s)$, or $s'' = d \cdot s'''$ and $s''' \geq_{\mathsf{obs}} s'$. Therefore, $[\![\mathsf{now\ c\ else\ P}]\!]_e(s') \uparrow$. In case $d \vdash c$ or $s = \epsilon$, then $s \in [\![\mathsf{now\ c\ else\ P}]\!]_e$ and this is against our assumptions.

Point 2 The monotonicity w.r.t. $\subseteq$ is obvious. We proceed in a similar way as in the previous case, thus assume that $s = d \cdot s' \in [\![\mathsf{now\ c\ then\ Q}]\!]_e \setminus [\![\mathsf{now\ c\ then\ P}]\!]_e$. If $d \vdash c$ then $s \in Q \setminus P$, and therefore $P(s) \uparrow$. As a result, for each $s' \geq_{\mathsf{obs}} s$, if $s' \in [\![\mathsf{now\ c\ then\ P}]\!]_e$ then $s' \geq_{\mathsf{obs}} P(s)$ and therefore $l(s') \leq l(s)$. Therefore, $[\![\mathsf{now\ c\ then\ P}]\!]_e(s) \uparrow$. In case $d \not\vdash c$ or $s = \epsilon$, then $s \in [\![\mathsf{now\ c\ then\ P'}]\!]_e$ and this is against our assumptions.

Point 3 The monotonicity w.r.t. $\subseteq$ is obvious. We assume $s = d \cdot s'$ in $[\![\mathsf{next\ Q}]\!]_e \setminus [\![\mathsf{next\ P}]\!]_e$. Then $s' \in Q \setminus P$ and therefore $P(s') \uparrow$. As an immediate consequence, $[\![\mathsf{next\ P}]\!]_e(s') \uparrow$.

Point 4 The monotonicity w.r.t. $\subseteq$ is obvious. If $s \in (Q_1 \cap Q_2) \setminus (P_1 \cap P_2)$ then wither $s \notin P_1$ or $s \notin P_2$. Assume without loss of generality that $s \notin P_1$. Then $P_1(s) \uparrow$, i.e., for each $s' \geq_{\mathsf{obs}} s$ with $s' \in P_1$, $l(s') < l(s)$. But this implies that for each $s' \geq s$ with $s' \in P_1 \cap P_2$, we have $l(s') < l(s)$, and therefore $(P_1 \cap P_2)(s) \uparrow$.

■

Thus, as we have proved that the semantics for each construct are monotonic, now we can say that the semantics of a program are continuous. This is what establishes Theorem 4.1.14.

Theorem 4.1.14 *If* A *is agent,* $[\![A]\!]$ *from* $(\mathbf{Env}, \leq_{\mathsf{div}})$ *to* $(\mathbf{Proc}, \leq_{\mathsf{div}})$ *is continuous.*

Proof. This is a direct consequence of Theorem 4.1.12 and Lemma 4.1.13. ■

Note that, as we anticipated before, this theorem does not hold if we replace $\leq_{\mathsf{div}}$ with $\subseteq$, since $\exists_x$ is not monotone w.r.t. $\subseteq$. For example consider $P = \{\epsilon, x = 0, x = 0 \cdot \mathit{true}\}$ and $Q = P \cup \{x = 0 \vee x = 1\}$. Then $\mathit{true} \cdot \mathit{true} \in \exists_x P$ since $\exists_x \bar{P}(\mathit{true} \cdot \mathit{true}) = \exists_x (x = 0 \cdot \mathit{true}) = \mathit{true} \cdot \mathit{true}$. However $\exists_x \bar{Q}(\mathit{true} \cdot \mathit{true}) = \exists_x (x = 0 \vee x = 1) = \mathit{true} \neq \mathit{true} \cdot \mathit{true}$. Therefore $\mathit{true} \cdot \mathit{true} \notin \exists_x Q$. Note that although $P \subseteq Q$, $x = 0 \vee x = 1 \notin d(P)$, since $\bar{P}(x = 0 \vee x = 1) = (x = 0)$. Therefore $P \not\leq_{\mathsf{div}} Q$.

4.2 Correctness and Completeness

In this section we show the correctness and completeness of the new denotational semantics. Thus, we show that the new denotational semantics defined is fully abstract w.r.t. the observable of strongest postconditions.

First of all, we define the strongest postconditions of a set of definitions. Given a set of definitions D, we denote with $sp(\mathsf{D})$ the environment such that

$$sp(\mathsf{D})(\mathsf{p}) = \begin{cases} sp(\mathsf{D}.\mathsf{p}(\vec{x}))[\vec{x}/\vec{\alpha}] & \text{if } \mathsf{p}(\vec{x}) :: \mathsf{A}_\mathsf{P} \in \mathsf{D} \\ \{\epsilon\} & \text{otherwise} \end{cases}$$

We also define the strongest postcondition of a configuration $\langle V, \Gamma \rangle$ in the obvious way, i.e.,

$$sp(\mathsf{D}.\langle V, \Gamma \rangle) = \{s \mid \exists \langle V', \Gamma' \rangle\ \exists s' \in \mathbf{Obs}.\ \langle V, \Gamma \rangle \overset{(s',s)}{\Longrightarrow}{}^{*} \langle V', \Gamma' \rangle\}$$

Given a configuration $\langle V, \Gamma \rangle$, we define a corresponding agent

$$a(\langle V, \Gamma \rangle) = \begin{cases} \exists_V A_1 \parallel \ldots \parallel A_n & \text{if } \Gamma = A_1 \uplus \ldots \uplus A_n \\ \mathsf{skip} & \text{if } \Gamma = \emptyset \end{cases} \tag{4.2.1}$$

Note that $a(\langle \emptyset, A \rangle) = A$.

The correctness of the denotational semantics is given by a theorem, but we first need to prove some lemma that is used in the proof of such theorem. The main intermediate result for the proof of correctness is Lemma 4.2.6 which relates the behavior of the operational semantics and the one of the denotational ones.

The first step towards correctness is Lemma 4.2.1. It tells us that, if a variable is not in the set of free variables of an agent Γ, then we can existentially quantify by such variable a constraint that appears concurrently to the agent Γ.

Lemma 4.2.1 *If* $\langle V, \Gamma \uplus c \rangle \longrightarrow \langle V', \Gamma' \uplus c \rangle$ *and* $x \notin fv(\Gamma)$, *then* $\langle V, \Gamma \uplus \exists_x c \rangle \longrightarrow \langle V', \Gamma' \uplus \exists_x c \rangle$.

Proof. If $x \notin fv(\Gamma)$ then $\Gamma = \exists_x \Gamma$. There are two possibilities, if $V = V'$ then we know that $x \notin fv(\Gamma')$, thus the theorem holds. In the other case, if $V \subset V'$ then we know that there is some y and $fv(\Gamma') = fv(\Gamma) \setminus \{y\}$ thus $x \notin fv(\Gamma')$ and the theorem also holds. ■

Lemma 4.2.2 allows us to define the parallel composition of two agents that have no shared variables as the union of its sets of variables and the union of multisets of agents.

Lemma 4.2.2 *If* $\langle V_1, \Gamma_1 \rangle \overset{(s,s)}{\Longrightarrow}{}^{*} \langle V_1', \Gamma_1' \rangle$ *and* $\langle V_2, \Gamma_2 \rangle \overset{(s,s)}{\Longrightarrow}{}^{*} \langle V_2', \Gamma_2' \rangle$ *with* $V_1' \cap V_2' = \emptyset$, $V_1' \cap fv(\Gamma_1) = V_2' \cap fv(\Gamma_2) = \emptyset$, *then* $\langle V_1 \cup V_2, \Gamma_1 \uplus \Gamma_2 \rangle \overset{(s,s)}{\Longrightarrow}{}^{*} \langle V_1' \cup V_2', \Gamma_1' \uplus \Gamma_2' \rangle$.

Proof. We proceed by induction on the length of s. If $s = \epsilon$, then $V_1' = V_1$, $\Gamma_1' = \Gamma_1$, $V_2' = V_2$, $\Gamma_2' = \Gamma_2$ and it trivially holds that $\langle V_1 \cup V_2, \Gamma_1 \uplus \Gamma_2 \rangle \stackrel{(\epsilon,\epsilon)}{\Longrightarrow}{}^* \langle V_1 \cup V_2, \Gamma_1 \uplus \Gamma_2 \rangle$. Now, if we assume that $s = c \cdot s'$, then there is a sequence

$$\langle V_1, \Gamma_1 \uplus \exists_{V_1} c \rangle \longrightarrow^* \langle W_1, \Delta_1 \uplus \exists_{V_1} c \rangle \rightsquigarrow \langle W_1, \Delta_1' \rangle$$

such that $\langle W_1, \Delta_1' \rangle \stackrel{(s',s')}{\Longrightarrow}{}^* \langle V_1', \Gamma_1' \rangle$, and a corresponding derivation for $\langle V_2, \Gamma_2 \rangle$. Since agents are well formed and by the Lemma 4.2.1, we can paste the two $\longrightarrow^*$ sequences together to obtain

$$\langle V_1, \Gamma_1 \uplus \Gamma_2 \uplus \exists_{V_1 \cup V_2} c \rangle \longrightarrow^* \langle W_1, \Delta_1 \uplus \exists_{V_1 \cup V_2} c \uplus \Gamma_2 \rangle \longrightarrow^* \langle W_1 \cup W_2, \Delta_1 \uplus \Delta_2 \uplus \exists_{V_1 \cup V_2} c \rangle$$

For such combined sequence, we need to prove that there is no any active agent in $\Delta_1 \uplus \Delta_2$. Assume that the agent $A \in \Delta_1$ is active, i.e.,

$$\langle W_1 \cup W_2, A \uplus \Delta_1' \uplus \Delta_2 \uplus \exists_{V_1 \cup V_2} c \rangle \longrightarrow \langle W, \Theta \uplus \Delta_1' \uplus \Delta_2 \uplus \exists_{V_1 \cup V_2} c \rangle$$

where $\Delta_1 = A \uplus \Delta_1'$. Now we prove that in that case, A is an active agent also for $\langle W_1, \Delta_1 \uplus \exists_{V_1} c \rangle$ dealing to a contradiction with the initial conditions. The only non trivial case is when $A = \mathsf{now}\ d\ \mathsf{then}\ B$ or $A = \mathsf{now}\ d\ \mathsf{else}\ B$. In the first case case, it holds that $\sigma(\Delta_1) \sqcup \sigma(\Delta_2) \sqcup \exists_{V_1 \cup V_2} c \vdash d$. Therefore, it is also the case that $\exists_{V_2} \sigma(\Delta_1) \sqcup \exists_{V_2} \sigma(\Delta_2) \sqcup \exists_{V_2 \cup V_1} c \vdash \exists_{V_2} d$. However, since agents are well formed, then $\exists_{V_2} d = d$, $\exists_{V_2} \sigma(\Delta_1) = \sigma(\Delta_1)$ and $c \sqcup \exists_{V_2} \sigma(\Delta_2) = c \sqcup \exists_{V_1} \sigma(\Delta_1)$. Henceforth, we have $\sigma(\Delta_1) \vdash d$ and therefore A is active in $\langle W_1, \Delta_1 \rangle$. As a result, we have

$$\langle W_1 \cup W_2, \Delta_1 \uplus \Delta_2 \uplus \exists_{V_1 \cup V_2} c \rangle \rightsquigarrow \Delta_1' \uplus \Delta_2'$$

and therefore

$$\langle V_1 \cup V_2, \Gamma_1 \uplus \Gamma_2 \rangle \stackrel{(c,c)}{\Longrightarrow} \langle W_1 \cup W_2, \Delta_1' \uplus \Delta_2' \rangle$$

By inductive hypothesis follows the thesis.

The case for $A = \mathsf{now}\ d\ \mathsf{else}\ B$ it holds that $\sigma(\Delta_1) \sqcup \sigma(\Delta_2) \sqcup \exists_{V_1 \cup V_2} c \nvdash d$ and following a similar idea, we proof the thesis. ■

The following two results associate the notion of postcondition and closure operator. The first one is relative to agents whereas the second one defines the substitution for postconditions.

Lemma 4.2.3 *If* $P = sp(D.A)$ *then* $\bar{P}(s) = s'$ *iff* $\langle \emptyset, A \rangle \stackrel{(s,s')}{\Longrightarrow}{}^*$.

Proof. The proof is by induction on the length of s and s'. When the length is zero the property is trivial. Otherwise, assume $\bar{P}(c \cdot s) = c' \cdot s'$, and we proceed by induction on the structure of A. It is not possible that $A = \mathsf{abort}$, otherwise $P = \{\epsilon\}$ and $\bar{P}(s) \uparrow$. If $A = \mathsf{skip}$ then $P = \mathbf{Obs}$ and therefore $\bar{P}(s) = s$ and $\langle \emptyset, A \rangle \stackrel{(s,s)}{\Longrightarrow}{}^*$ for each $s \in \mathbf{Obs}$.

■

Lemma 4.2.4 $sp(\mathsf{D}.\mathsf{p}(\vec{y})) = sp(\mathsf{D}.\mathsf{p}(\vec{x}))[\vec{x}/\vec{y}]$.

Proof. The proof is by induction on the length of the derivation. We start by proving $sp(\mathsf{D}.\mathsf{p}(\vec{y})) \subseteq sp(\mathsf{D}.\mathsf{p}(\vec{x}))[\vec{x}/\vec{y}]$. Given $s \in sp(\mathsf{D}.\mathsf{p}(\vec{y}))$, the case $s = \epsilon$ is trivial. Note that $\langle\emptyset, \mathsf{p}(\vec{y})\rangle \longrightarrow \langle\emptyset, A_\mathsf{p}[\vec{z}/\vec{y}]\rangle$ while $\langle\emptyset, \mathsf{p}(\vec{x})\rangle \longrightarrow \langle\emptyset, A_\mathsf{p}[\vec{z}/\vec{x}]\rangle$. By the Theorem 3.2.3, since $A_\mathsf{p}[\vec{z}/\vec{y}] = A_\mathsf{p}[\vec{z}/\vec{x}][\vec{x}/\vec{y}]$, we have $s \in sp(\mathsf{D}.\mathsf{p}(\vec{y})) = sp(\mathsf{D}.\mathsf{p}(\vec{x}))[\vec{x}/\vec{y}]$. ■

In Lemma 4.2.5 we show that the introduction of a fresh variable in the configuration which is not contained in the set of variables of the program does not introduce new transitions.

Lemma 4.2.5 *If* $\langle V, \Gamma\rangle \overset{(c,c')}{\Longrightarrow} \langle W, \Delta\rangle$ *with* $x \notin W$, $x \notin fv(c)$, *then* $\langle V \cup x, \Gamma\rangle \overset{(c,\exists_x c')}{\Longrightarrow} \langle W \cup x, \Delta\rangle$.

Proof. It is trivial to check that if $\langle V, \Gamma\rangle \longrightarrow \langle W, \Delta\rangle$ with $x \notin W$, then $\langle V \cup x, \Gamma\rangle \longrightarrow \langle W \cup x, \Delta\rangle$. The same holds for the relation $\rightsquigarrow$. If $\langle V, \Gamma\rangle \overset{(c,c')}{\Longrightarrow} \langle W, \Delta\rangle$ it means that

$$\langle V, \Gamma \uplus \exists_V c\rangle \longrightarrow \langle W, \Gamma'\rangle \rightsquigarrow \Delta.$$

where $c' = \exists_V c \sqcup \exists_W \sigma(\Delta)$. Since $x \notin fv(c)$ then $\exists_V c = \exists_{V\cup x} c$ and therefore we have

$$\langle V \cup x, \Gamma \uplus \exists_{V\cup x} c\rangle \longrightarrow \langle W \cup x, \Gamma'\rangle \rightsquigarrow \Delta.$$

which implies

$$\langle V \cup x, \Gamma\rangle \overset{(c,\exists_V c \sqcup \exists_{W\cup x}\sigma(\Delta))}{\Longrightarrow} \langle W \cup x, \Delta\rangle$$

But since $\exists_V c \sqcup \exists_{W\cup x} = \exists_x(\exists_V c \sqcup \exists_W c = \exists_x c')$ we have the thesis. ■

Now we present the main lemma for the proof of correctness. The Lemma 4.2.6 says that the denotational semantics of an agent A where the environment e is the strongest postconditions of a set of definitions, corresponds with the postconditions of the program D.A composed by such set of definitions and the agent A. Moreover, if we use the closure operator, we have the input-output behavior of the program D.A.

Lemma 4.2.6 *If* $e = sp(\mathsf{D})$ *then* $[\![a(\langle V, \Gamma\rangle)]\!]_e = sp(\mathsf{D}.\langle V, \Gamma\rangle)$.

Proof. First of all, note that the only transition starting from $\langle V, \emptyset\rangle$ are those of the form

$$\langle V, \emptyset\rangle \overset{(c,c)}{\Longrightarrow} \langle V, \emptyset\rangle$$

Now, the proof proceed by structural inductions on the agent A:

$\underline{\mathbf{c}|}$ $\langle\emptyset, c\rangle \overset{(d,d\sqcup c)}{\Longrightarrow} \langle\emptyset, \emptyset\rangle$ is the only possible $\Longrightarrow$-transition. Given the previous observation, $sp(\mathsf{D}.c) = \{d \sqcup c \cdot s \mid d \in \mathcal{C}, s \in \mathbf{Obs}\} = [\![c]\!]_e$.

skip The only possible transition starting from $\langle\emptyset, \mathsf{skip}\rangle$ is $\langle\emptyset, \mathsf{skip}\rangle \stackrel{(c,c)}{\Longrightarrow} \langle\emptyset,\emptyset\rangle$ for every $c \in \mathcal{C}$. Therefore $sp(D.\,\mathsf{skip}) = \mathbf{Obs} = [\![\mathsf{skip}]\!]_e$.

abort $\langle\emptyset, \mathsf{abort}\rangle \longrightarrow \langle\emptyset, \mathsf{abort}\rangle$ is the only possible transition from $\langle\emptyset, \mathsf{abort}\rangle$. Therefore, there is no $\Longrightarrow$ transition starting from the same configuration, which means that $sp(D.\,\mathsf{abort}) = \{\epsilon\} = [\![\mathsf{abort}]\!]_e$.

$\mathsf{now}\ c\ \mathsf{else}\ A_1$ Assume that we have a specific derivation starting from the configuration $\langle\emptyset, \mathsf{now}\ c\ \mathsf{else}\ A_1\rangle$, i.e., $\langle\emptyset, \mathsf{now}\ c\ \mathsf{else}\ A_1\rangle \stackrel{(d,d)}{\Longrightarrow} \langle\emptyset, \Delta\rangle$. There are two cases, i.e., either $d \vdash c$ and therefore there is an active agent in the starting configuration, or $d \nvdash c$. In the first case,

$$\langle\emptyset, \mathsf{now}\ c\ \mathsf{else}\ A_1\rangle \stackrel{(d,d)}{\Longrightarrow} \langle\emptyset,\emptyset\rangle$$

while in the latter

$$\langle\emptyset, \mathsf{now}\ c\ \mathsf{else}\ A_1\rangle \stackrel{(d,d)}{\Longrightarrow} \langle\emptyset, A_1\rangle$$

Since by inductive hypothesis, $[\![A_1]\!]_e = sp(D.A_1)$, we have that $[\![\mathsf{now}\ c\ \mathsf{else}\ A_1]\!]_e = sp(D.\,\mathsf{now}\ c\ \mathsf{else}\ A_1)$ by definition of $[\![\_]\!]_e$.

$\mathsf{now}\ c\ \mathsf{then}\ A_1$ Assume that we have a specific derivation starting from the configuration $\langle\emptyset, \mathsf{now}\ c\ \mathsf{then}\ A_1\rangle$, i.e., $\langle\emptyset, \mathsf{now}\ c\ \mathsf{then}\ A_1\rangle \stackrel{(d,d)}{\Longrightarrow} \langle W, \Delta\rangle$. There are two cases, i.e., either $d \vdash c$ and therefore there is an active agent in the starting configuration, or $d \nvdash c$. In the first case,

$$\langle\emptyset, d \uplus \mathsf{now}\ c\ \mathsf{else}\ A_1\rangle \longrightarrow \langle\emptyset, d \uplus A_1\rangle \longrightarrow^* \langle W, \Delta'\rangle \rightsquigarrow \Delta$$

This mean that also $\langle\emptyset, A_1\rangle \stackrel{(d,d)}{\Longrightarrow} \langle W, \Delta\rangle$. This means that $d \cdot s \in sp(D.A_1)$. Moreover, since the first $\longrightarrow$ transition from $\langle\emptyset, d \uplus \mathsf{now}\ c\ \mathsf{else}\ A_1\rangle$ is forced, the converse is also true, i.e., if $d \cdot s \in sp(D.A_1)$, then $d \cdot s \in sp(D.\,\mathsf{now}\ c\ \mathsf{then}\ A_1)$.

In the second case, i.e., $d \nvdash c$, we have

$$\langle\emptyset, d \uplus \mathsf{now}\ c\ \mathsf{else}\ A_1\rangle \stackrel{(d,d)}{\Longrightarrow} \langle\emptyset,\emptyset\rangle$$

and therefore $d \cdots s \in sp(D.\,\mathsf{now}\ c\ \mathsf{else}\ A_1)$ for any $s \in \mathbf{Obs}$. By the definition of $[\![\_]\!]_e$, we have $[\![\mathsf{now}\ c\ \mathsf{else}\ A_1]\!]_e = sp(D.\,\mathsf{now}\ c\ \mathsf{else}\ A_1)$.

$\mathsf{next}\ A_1$ The only derivation starting from $\langle\emptyset, \mathsf{next}\ A_1\rangle$ is

$$\langle\emptyset, \mathsf{next}\ A_1\rangle \stackrel{(c,c)}{\Longrightarrow} \langle\emptyset, A_1\rangle$$

for any $c \in \mathcal{C}$. Therefore, it follows $sp(D.\,\mathsf{next}\ A_1) = [\![\mathsf{next}\ A_1]\!]_e$.

$p(\vec{x}))$ Assume $p(\vec{y}) :: A_P \in D$. Then $e(p) = sp(D.p(\vec{\alpha}))$ and

$$[\![p(\vec{y})]\!]_e = \exists_{\vec{\alpha}}([\![\vec{y} = \vec{\alpha}]\!]_e \cap sp(p(\vec{\alpha})))$$

while $sp(D.p(\vec{y})) = sp(D.p(\vec{x}))[x/y]$.

$\underline{\mathbf{A_1 \| A_2}}\rfloor$ If $s \in [\![A_1 \| A_2]\!]_e$ then $s \in [\![A_1]\!]_e$ and $s \in [\![A_2]\!]_e$. By inductive hypothesis, $s \in sp(D.A_1) \cap sp(D.A_2)$. We have two sequences $\xi_1 : \langle \emptyset, A_1 \rangle \overset{(s,s)}{\Longrightarrow} \langle V_1, \Gamma_1 \rangle$ and $\xi_2 : \langle \emptyset, A_2 \rangle \overset{(s,s)}{\Longrightarrow} \langle V_2, \Gamma_2 \rangle$ and by Theorem 3.2.3 (on renaming) we can choose V_1 and V_2 to be disjoint and renamed apart from the free variables in Γ_1 and Γ_2. By the Lemma 4.2.2 we have

$$\langle \emptyset, A_1 \| A_2 \rangle \longrightarrow \langle \emptyset, A_1 \uplus A_2 \rangle \overset{(s,s)}{\Longrightarrow}{}^{*} \langle V_1 \cup V_2, \Gamma_1 \uplus \Gamma_2 \rangle$$

and this proves that $[\![A]\!]_e \subseteq sp(D.A)$. For the converse inequality, note that if $s \in sp(D.A)$ then $s \in sp(D.A_1)$ and $s \in sp(D.A_2)$ and therefore

$\underline{\mathbf{x^\frown A_1}}\rfloor$ If $s \in [\![x^\frown A_1]\!]_e$, then $\exists_x s = \exists_x \bar{P}(\exists_x s)$ where $P = [\![A_1]\!]_e$. By inductive hypothesis, if $s' = \bar{P}(\exists_x s)$, then $\langle \emptyset, A_1 \rangle \overset{(\exists_x s, s')}{\Longrightarrow}$. Therefore, since $\langle \emptyset, A \sqcup \exists_x s \rangle \longrightarrow \langle y, A_1[x/y] \sqcup \exists_x s \rangle$ for y fresh, then we know that $\langle x, A_1 \rangle \overset{(\exists_x s, \exists_x s')}{\Longrightarrow}{}_{*}$ and the same holds for $\langle \emptyset, A \rangle$ too.

On the converse, assume $\langle \emptyset, A \rangle \overset{(s,s')}{\Longrightarrow}{}^{*}$. Then, $\langle x, A_1 \rangle \overset{(\exists_x s, s'')}{\Longrightarrow}$ such that $s' = \exists_x s''$. If $s' = s$ then $\exists_x s'' = s' = s$ and therefore $\exists_x s'' = \exists_x s$. This means $s' \in [\![A]\!]_e$.

■

Thanks to this last result and the fact that all the operators are monotonic w.r.t. $\leq_{\mathsf{div}}$ (see Theorem 4.1.14), we obtain the required result of correctness.

Theorem 4.2.7 *For all the programs* $D.A$, $[\![D.A]\!] \leq sp(D.A)$.

Proof. This is a direct consequence from the Lemma 4.2.6 and Theorem 4.1.14 ■

For the proof of completeness, the following lemmata prove that operational steps may be simulated in the denotational semantics. First of all we show three intermediate results. In Lemma 4.2.8 it is proved that, given a transition of the transition relation $\longrightarrow$, the denotations corresponding to the target configuration are included in the denotations of the source agent.

Lemma 4.2.8 *Given a set of declarations* D, *if* $\langle V, A \rangle \longrightarrow \langle V', A' \rangle$ *then* $[\![D.\exists_{V'} A']\!] \subseteq [\![D.\exists_V A]\!]$.

Proof. The proof is by induction on the structure of A. Here we show the most interesting cases, i.e., the cases for the Parallel and the Procedure Agent constructs.

$\underline{\mathbf{abort}}\rfloor$. Then $\langle V, A \uplus \mathsf{abort} \rangle \longrightarrow \langle V, \mathsf{abort} \rangle$. The result is a direct consequence of the fact that $[\![A \uplus \mathsf{abort}]\!] = \emptyset = [\![\mathsf{abort}]\!]$.

$\underline{\mathbf{A_1 \| A_2}}\rfloor$. Trivial.

$\underline{\mathbf{p(\vec{x})}}\rfloor$. Then $\langle V, A \uplus p(\vec{y}) \rangle \longrightarrow \langle V, A \uplus A_p[\vec{x}/\vec{y}]$. Note that $[\![p(\vec{y})]\!]_e = e(p)[\vec{\alpha}/\vec{y}]$. Since $e = \mathit{lfp}[\![D]\!]$, then $e(p) = [\![A_p]\!]_e[\vec{x}/\vec{\alpha}]$ and therefore $[\![p(\vec{y})]\!] = [\![A_p]\!]_e[\vec{x}/\vec{\alpha}][\vec{\alpha}/\vec{y}] = [\![A_p]\!]_e[\vec{x}/\vec{y}]$.

■

Secondly, in Lemma 4.2.9 we show that for the transition relation which describes the passage of time, the denotations corresponding to the source configuration and its store are included in the denotations of the target configuration.

Lemma 4.2.9 *Given a program* $\mathsf{D}.\mathsf{A}$*, if* $\langle \mathsf{V}, \mathsf{A}\rangle \rightsquigarrow \langle \mathsf{V}, \mathsf{A}'\rangle$ *then,* $[\![\mathsf{D}.\exists_{\mathsf{V}'}\mathsf{A}']\!] \supseteq \sigma(\mathsf{A}) \cdot [\![\mathsf{D}.\exists_{\mathsf{V}}\mathsf{A}]\!]$.

Finally, in Lemma 4.2.10 we show that for each observable s in the denotations of the target agent, the observable for the source configuration is defined as the strong-postcondition and the observable s

Lemma 4.2.10 *Given a program* $\mathsf{D}.\mathsf{A}$*, if* $\langle \mathsf{V}, \mathsf{A}\rangle \overset{(\mathsf{c},\mathsf{c})}{\Longrightarrow} \langle \mathsf{V}', \mathsf{A}'\rangle$ *then it is true that for each* $\mathsf{s} \in [\![\mathsf{D}.\exists_{\mathsf{V}'}\mathsf{A}']\!]$*,* $\mathsf{c} \cdot \mathsf{s} \in [\![\mathsf{D}.\exists_{\mathsf{V}}\mathsf{A}]\!]$.

Proof. If $\langle \mathsf{V}, \mathsf{A}\rangle \overset{(\mathsf{c},\mathsf{c})}{\Longrightarrow} \langle \mathsf{V}', \mathsf{A}'\rangle$ then $\langle \mathsf{V}, \mathsf{A} \uplus \exists_{\mathsf{V}}\mathsf{c}\rangle \longrightarrow^* \langle \mathsf{V}', \mathsf{A}''\rangle \rightsquigarrow \langle \mathsf{V}', \mathsf{A}'\rangle$ with $\mathsf{c} \sqcup \exists_{\mathsf{V}'}\sigma(\mathsf{A}'') = \mathsf{c}$, i.e., $\exists_{\mathsf{V}'}\sigma(\mathsf{A}'') \sqsubseteq \mathsf{c}$. Given $\mathsf{s} \in [\![\mathsf{D}.\exists_{\mathsf{V}'}\mathsf{A}']\!]$, by Lemma 4.2.9, ■

As a direct consequence of the previous lemmata, we have the following result.

Lemma 4.2.11 *Given a set of definitions* D*, if* $\langle \mathsf{V}, \mathsf{A}\rangle \overset{(\mathsf{s},\mathsf{s})}{\Longrightarrow}{}^* \langle \mathsf{V}', \mathsf{A}'\rangle$*, then for each* $\mathsf{s}' \in [\![\mathsf{D}.a(\langle \mathsf{V}', \mathsf{A}'\rangle)]\!]$*,* $\mathsf{s} \cdot \mathsf{s}' \in [\![\mathsf{D}.a(\langle \mathsf{V}, \mathsf{A}\rangle)]\!]$.

Proof. The proof is by induction on the length of s. If $\mathsf{s} = \epsilon$ then $\mathsf{V}' = \mathsf{V}$ and $\mathsf{A}' = \mathsf{A}$ and the result is trivial. Otherwise $\mathsf{s} = \mathsf{c} \cdot \mathsf{s}'$ and $\langle \mathsf{V}, \mathsf{A}\rangle \overset{(\mathsf{c},\mathsf{c})}{\Longrightarrow} \langle \mathsf{V}'', \mathsf{A}''\rangle \overset{(\mathsf{s}',\mathsf{s}')}{\Longrightarrow}{}^* \langle \mathsf{V}', \mathsf{A}'\rangle$. By inductive hypothesis, for each $\mathsf{s}'' \in [\![\mathsf{D}.\exists_{\mathsf{V}'}\mathsf{A}']\!]$, $\mathsf{s}' \cdot \mathsf{s}'' \in \mathsf{sem}\mathsf{D}.\exists_{\mathsf{V}''}\mathsf{A}''$. By Lemma 4.2.10, this implies $\mathsf{s} \cdot \mathsf{s}'' = \mathsf{c} \cdot \mathsf{s}' \cdot \mathsf{s}'' \in [\![\mathsf{D}.\exists_{\mathsf{V}}\mathsf{A}]\!]$ and this proves the thesis. ■

Therefore, now we can prove that the strongest-postconditions of given program are included in its denotational semantics.

Theorem 4.2.12 *For all the programs* $\mathsf{D}.\mathsf{A}$*,* $sp(\mathsf{D}.\mathsf{A}) \subseteq [\![\mathsf{D}.\mathsf{A}]\!]$.

Proof. This is a direct consequence from Lemma 4.2.11. ■

Finally, by Theorems 4.2.12 and 4.2.7 we have the final result

Corollary 4.2.13 *Given a program* $\mathsf{D}.\mathsf{A}$*, we have* $sp(\mathsf{D}.\mathsf{A}) = [\![\mathsf{A}]\!]$ *and* $io(\mathsf{D}.\mathsf{A}) = [\![\mathsf{A}]\!]$.

This states that the denotational semantics is fully-abstract w.r.t. strongest postconditions and input-output observables.

4.2.1 On the Expressive Power of tcc

The language tcc is derived from deterministic concurrent constraint programming model by adding time and detection of negative information. It is interesting to examine the impact of these extensions on the expressive power of the language. In this section we present an initial analysis, focusing on the impact of the $\mathsf{now}\ c\ \mathsf{else}\ A$ construct.

Therefore, let us consider tcc without the $\mathsf{now}\ c\ \mathsf{else}\ A$ operator. In this case, the agent which is executed at time $n+1$ does not depend from the negative information discovered at time n.

We may extend $\sqsubseteq$ pointwise on observations, by defining

$$c_1 \dots c_n \sqsubseteq d_1 \dots d_m \iff m = n \wedge \forall 1 \leq i \leq m.\ c_i \sqsubseteq d_i \tag{4.2.2}$$

Then, we can formalize the idea that a process does not use negative information by the following result

Definition 4.2.14 *A process* P *is* monotonic *when, given* $s, s' \in P$*, if* $s \sqsubseteq s'$ *then* $P \,\mathsf{after}\, s \supseteq P \,\mathsf{after}\, s'$.

Note that this is not always the case in the general tcc. For example, consider the process

$$P = [\![\mathsf{now}\ c\ \mathsf{else}\ c]\!]_e$$

for $c \neq \mathit{true}$ and an environment e. Then $\mathit{true} \sqsubseteq c$ but $\mathit{true} \in P \,\mathsf{after}\ c$ while $\mathit{true} \notin P \,\mathsf{after}\, \mathit{true}$. We may prove that in tcc without $\mathsf{now}\ c\ \mathsf{else}\ A$, all the processes are monotonic.

Theorem 4.2.15 *Let* A *be an agent and* D *a set of declarations built without the operator* $\mathsf{now}\ c\ \mathsf{else}\ A$. *Then* $[\![D.A]\!]$ *is a monotonic process.*

There is also an alternative characterization of monotonicity. This requires to use the $\leq_{\mathsf{obs}}$ ordering presented in (3.2.3). This is a sort of pointwise extension of entailment between constraints, with the proviso that an aborted process is like a process which always emits *false*, the greatest element of $\mathcal{C}$.

It is the case that $(\mathbf{Obs}, \leq_{\mathsf{obs}})$ is a complete sup-semilattice without least element. Given a process P, $\bar{P}$ is extensive and idempotent over $(\mathbf{Obs}, \leq_{\mathsf{obs}})$. In general, however, it is not monotone. We can prove the following theorem:

Theorem 4.2.16 *If* P *is monotonic then* $\bar{P}$ *is monotonic (hence it is a closure operator) over* $(\mathbf{Obs}, \leq_{\mathsf{obs}})$.

Therefore a monotone tcc process may be viewed as a cc agent in an extended constraint system which includes a notion of time.

Another interesting thing of monotonic processes is that if P is monotonic then $\exists_x P = \{s \mid \exists_x s = \exists_x s' \wedge s' \in P\}$. This is indeed the definition given in [SJG94a, NPV02b], and it makes for a further justification of the fact that the denotational semantics of [NPV02b] corresponds to the operational semantics in the case of locally independent processes.

4.3 Related Works

Now, we come back to the original semantics for tcc presented in [SJG94a]. Apart from the absence of environments, omitted to simplify the presentation, the major difference with our semantics is in the definition of the operator $\exists_x$:

$$\exists_x P = \{s \in \mathbf{Obs} \mid \exists_x s = \exists_x t \text{ for some } t \in P\} \tag{4.3.1}$$

As we have said before, this is essentially the same definition which appears in [SRP91], just extended over time. However, in [SRP91] there was no way of detecting negative information. Now that we have this feature, the use of $\exists_x$ as in (4.3.1) gives a denotational semantics which does not correspond to the operational one, but only approximate it from the above.

In the rest of this section we denote with $\exists_x$ our definition of the hiding operator and with $\tilde{\exists}_x$ the one given in (4.3.1). We also use $[\![\ ]\!]$ and $\{\!\{\ \}\!\}$ for the corresponding denotational semantics.

Let us consider the agent

$$A = x\hat{\ }(\mathsf{now}\ x = a\ \mathsf{else}\ y = b)$$

We omit environment from the semantic braces since we do not have definitions. We have that $\{\!\{A\}\!\} = \{\!\{\mathsf{skip}\}\!\}$. Actually, for each $s \in \mathbf{Obs}$, if $s = \epsilon$ then $s \in \{\!\{\mathsf{now}\ x = \tilde{a}\ \mathsf{else}\ y = b\}\!\}$ by definition. Otherwise, $s = d \cdot s'$ and consider $s'' = (\exists x.(d \sqcup x = a)) \cdot s'$. Then $\exists x.s = \exists x.d \cdot \exists x.s' = \exists x.s''$ and $s'' \in \{\!\{\mathsf{now}\ x = a\ \mathsf{else}\ y = b\}\!\}$. Therefore $s \in \{\!\{A\}\!\}$. But skip and A are not observationally equivalent. Actually,

$$\langle \emptyset, A\rangle \overset{(\mathit{true},\mathit{true})}{\Longrightarrow} \langle \{x'\}, y = b\rangle \overset{(\mathit{true},y=b)}{\Longrightarrow} \langle \emptyset, \emptyset\rangle$$

while

$$\langle \emptyset, \mathsf{skip}\rangle \overset{(\mathit{true},\mathit{true})}{\Longrightarrow} \langle \emptyset, \emptyset\rangle \overset{(\mathit{true},\mathit{true})}{\Longrightarrow} \langle \emptyset, \emptyset\rangle$$

Therefore $\mathit{true} \cdot \mathit{true} \in \mathit{sp}(\mathsf{skip})$ while $\mathit{true} \cdot \mathit{true} \notin \mathit{sp}(A)$.

Note that [SJG94b] actually defines a slightly different notion of input-output behavior than ours. The observations of a program G_1 such that $\mathit{fv}(G_1) = V$ for the input sequence $(c_1, \ldots, c_n)$ is

$$O(G_1)(c_1, \ldots, c_n) \overset{d}{=} (\delta V.d_1, \ldots, \delta V.d_n),$$

$$\text{if } G_i \parallel c_i \longrightarrow R_{i+1}, \sigma(R_i) = d_i, R_i \rightsquigarrow G_{i+1}, 1 \leq i < n$$

where $\delta_V.c$ quantifies over all the variables which are not in V. But again, according to this definition, we have

$$O(x\hat{\ }(\mathsf{now}\ x = a\ \mathsf{else}\ y = b))(\mathit{true}) = (\delta y.y = b) = (y = b)$$

and

$$O(\mathsf{skip})(\mathit{true}) = (\delta x.\mathit{true}) = (\mathit{true})$$

Other semantics for timed cc languages may be found in [NPV02b, BGM00]. In [NPV02b] the language ntcc is introduced, which is a non-deterministic extension of tcc. The semantics is similar to that of [SJG94a], and it is always based on the strongest postconditions. As a consequence, it has the same problem of tcc w.r.t. the bad interaction between the hiding operator and the detection of negative information. However, due to the presence of non-determinism, we do not think that our idea can be applied to ntcc directly. In [BGM00] a new language is presented, but based on a different point of view, since the passing of time is implicit at every transition step, and not explicitly introduced by syntactic constructs. The authors provide both an operational and a denotational semantics, which are proved to be equal. Moreover, the language is monotonic, thus it is quite different in nature from tcc.

In [BKPR92] it is presented a general construct for locality in languages based on asynchronous communication. An uniform semantic description of local variables (in imperative languages) and hiding of logical variables (in concurrent constraint languages) is introduced. The semantics defined for tccp follows the idea presented in such work. The problem to apply such kind of semantics to the tcc language comes from the non monotonicity of tcc.

In particular, in [BKPR92] the correctness of the denotational semantics depends on the fact that sequences of stores are connected. Under these conditions of monotonicity, the denotational semantics are fully abstract. Sequences of tcc are not connected, thus the uniform framework described in [BKPR92] cannot be applied to this language. We have shown that a possible solution is the semantics presented in this thesis.

5

The tccp language

The second programming language considered in this thesis is the *timed concurrent constraint programming* (tccp) language that was developed in [BGM00] by F. de Boer *et al.*. It was designed as a framework which allows one to model reactive and real-time systems. Thus, it is possible to specify and to verify distributed, concurrent systems. For these systems, the notion of time is a crucial question. tccp is based on the cc paradigm [Sar89, SR90, SRP91] that was presented as a general concurrent computational model. In [BGM00] the authors added the time concept to the cc paradigm obtaining an extension similar to the language of the approach presented in Chapter 3.

In cc there are some agents that add information into a store or check whether a constraint is entailed by the store. Thus, computations evolve as an accumulation of information into a global store. In tccp the agents defined for cc are inherited and it is introduced a *discrete global clock*. It is assumed that ask and tell actions take one time-unit and the parallel operator is interpreted in terms of maximal parallelism. Thus, computation evolves in steps of one time-unit. Another question is the time response of the constraint solver: it is assumed that a consult to the constraint solver takes a constant time, independently of the size of the store. In practice some restrictions (mentioned below) are taken in order to ensure that these hypothesis are reasonable (the reader can see [BGM00] for details).

To model reactive systems it is necessary to have the ability for describing notions as *timeout* or *preemption*. In tccp, this capability is introduced with the definition of a new operator (with respect to cc)

$$\mathsf{now}\ \mathsf{c}\ \mathsf{then}\ A\ \mathsf{else}\ \mathsf{B}$$

which tests if, in the current time instant, the store entails the constraint c and if it occurs, then in the same time instant it executes A; otherwise, it executes B (in the same time instant). It is necessary to fix a limit for the number of nested agents of this kind in order to ensure the bounded time response of the constraint solver. For recursive programs, such limit is ensured by the presence of the procedure call, since we assume that the evaluation of such a call takes one unit of time.

5.1 Syntax

The tccp language is parametric to an underlying constraint system. Thus, since now we assume that $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}, \mathcal{V}, \exists \rangle$ is the underlying constraint system for tccp where $\langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false} \rangle$ is a complete algebraic lattice, $\sqcup$ is the lub operation, and *true*, *false* are the least and the greatest elements of $\mathcal{C}$, respectively. Moreover, $\mathcal{V}$ is a (denumerable) set of variables with typical elements $x, y, z \ldots$. Finally, given $x \in \mathcal{V}$, $\exists_x : \mathcal{C} \rightarrow \mathcal{C}$ is the cylindrification operator (see Definition 1.4.2).

Then, given a cylindric constraint system $\mathcal{C}$, we show the syntax of the agents of the language in Figure 5.1. We assume that c and c_i are finite constraints (i.e., algebraic elements) in $\mathcal{C}$. Moreover, a tccp *process* is an object of the form D.A where D is a set of *procedure declarations* of the form $p(x) :- A$ and A is an agent.

(Agents)	$A ::=$	$\mathsf{tell}(c)$	– Tell
	$\mid$	abort	– Stop
	$\mid$	$\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$	– Choice
	$\mid$	$\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$	– Conditional
	$\mid$	$A \parallel B$	– Parallel
	$\mid$	$\exists x\, A$	– Hiding
	$\mid$	$p(x)$	– Procedure Call

Figure 5.1: tccp syntax [BGM00]

The Parallel, Hiding and Procedure Call agents are inherited from the cc model and behave in the same way. Thus, the Parallel agent represents the concurrency of the model and the Hiding operator makes a variable local to some process. The Procedure Call behaves as usually.

Now we consider the rest of the constructs introduced in Figure 5.1. We can observe two additional agents which were present in the cc model, but here these agents have a different semantics. In tccp, these two agents cause extension over time. The Tell agent adds the information c to the store, but this information is able to other agents only in the following time instant. This means that the tell action takes one unit of time. The same thing occurs with the Choice agent. Thus, we can say that the ask action takes also one unit of time since when we execute the $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ agent, we cannot start the execution of the A_i since the next time instant is reached.

Finally, the Conditional agent is the new agent introduced in the model in order to capture negative information. It behaves in a single instant of time in the sense that the condition is checked and *in the same instant of time* the execution of the corresponding agent is started. In particular, if the guard is satisfied, then A will be executed, otherwise the agent B will be executed. If we have two nested conditional agents, then the guards are recursively checked within the same time instant. This is the reason why we need a restriction about the maximum number of nested conditional agents.

5.2 Operational Semantics

In Figure 5.2 it is shown the operational semantics for tccp as described in [BGM00]. Each transition step takes one unit of time. In a configuration there are two components: a set of agents and a constraint representing the store. Therefore, the transition relation $\longrightarrow \subseteq \mathbf{Conf} \times \mathbf{Conf}$ is the least relation that satisfies the rules in Figure 5.2. We can say that the transition relation characterizes the (temporal) evolution of the system.

R1	$\langle \mathsf{tell}(c), d\rangle \longrightarrow \langle \mathsf{abort}, c \sqcup d\rangle$	
R2	$\langle \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle A_j, d\rangle$	$j \in [1,n]$ and $d \vdash c_j$
R3	$\dfrac{\langle A,d\rangle\longrightarrow\langle A',d'\rangle}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,d\rangle\longrightarrow\langle A',d'\rangle}$	$d \vdash c$
R4	$\dfrac{\langle A,d\rangle\not\longrightarrow}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,d\rangle\longrightarrow\langle A,d\rangle}$	$d \vdash c$
R5	$\dfrac{\langle B,d\rangle\longrightarrow\langle B',d'\rangle}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,d\rangle\longrightarrow\langle B',d'\rangle}$	$d \not\vdash c$
R6	$\dfrac{\langle B,d\rangle\not\longrightarrow}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,d\rangle\longrightarrow\langle B,d\rangle}$	$d \not\vdash c$
R7	$\dfrac{\langle A,c\rangle\longrightarrow\langle A',c'\rangle \quad \langle B,c\rangle\longrightarrow\langle B',d'\rangle}{\langle A \parallel B,c\rangle\longrightarrow\langle A' \parallel B',c'\sqcup d'\rangle}$	
R8	$\dfrac{\langle A,c\rangle\longrightarrow\langle A',c'\rangle \quad \langle B,c\rangle\not\longrightarrow}{\langle A \parallel B,c\rangle\longrightarrow\langle A' \parallel B,c'\rangle}$	
R9	$\dfrac{\langle A,c\rangle\longrightarrow\langle A',c'\rangle \quad \langle B,c\rangle\not\longrightarrow}{\langle A \parallel A,c\rangle\longrightarrow\langle B \parallel A',c'\rangle}$	
R10	$\dfrac{\langle A,d\sqcup\exists_x c\rangle\longrightarrow\langle B,d'\rangle}{\langle\exists^{d} x A,c\rangle\longrightarrow\langle\exists^{d'} x B,c\sqcup\exists_x d'\rangle}$	
R11	$\langle p(x), c\rangle \longrightarrow \langle A, c\rangle$	$p(x) : -A \in D$

Figure 5.2: Operational semantics for tccp language extracted from [BGM00]

Note that, differently from the tcc language, in tccp there is only one transition relation in the operational semantics. This makes easier to underestand the operational behaviour of the language. Since tccp considers the *maximal parallelism* mechanism in order to deal with concurrency, we must assume that there are as many processors as needed to execute a program. This behavior is described by means of rules **R7**, **R8** and **R9**. In these rules, the reader can see that whenever it is possible, we execute two agents concurrently. Otherwise we execute only one.

Rules **R3**, **R4**, **R5** and **R6** describe the operational semantics for the conditional agent. As the reader can observe, it depends on the store and on the initial configuration. Rule **R10** shows the semantics for the Hiding operator. Intuitively, the rule says that, if there exists a transition $\langle A, d \sqcup \exists_x c\rangle \longrightarrow \langle B, d'\rangle$, then d' is the local information produced by A. Moreover, the rule says that this local information d' must be hidden from the main process.

5.3 Applications

Using the basic constructs presented in Figure 5.1 it is possible to define other derived constructs. Such constructs make easier to the user to use the language. They make more intuitive the specification of systems which behave as timeouts or watchdogs. For example, in [BGM00] it is introduced a construct that behaves as the Choice agent since a timeout is reached:

$$\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i\, \mathsf{time\text{-}out}(m)\ B$$

It says that, before the time limit m is reached the process behaves in the same way as the Choice construct. However, after waiting for m time units, if no guard is enabled, then this agent behaves as B.

Another additional primitive was presented in [BGM00] by using the basic constructs. The watchdog agent is the typical preemption primitive and it is used to interrupt the activity of a process when some signal is presented.

$$\mathsf{do}\ A\ \mathsf{watching}\ \mathsf{c}$$

The reader can find more details about the semantics of this agent (and also about the semantics of the previous one) in the original work [BGM00]. Here we only have shown their syntax.

It is possible to find in the literature different examples of systems that can be modelled using the tccp language. As a first example, it is possible to model a system controller which checks that a acknowledgment signal arrives at most each ten time units. If it does not occur, then a recovery process is launched. Another clear example is the one which models the railroad crossing problem. This is a very typical problem of critical system which commonly appears in the literature (for example in [MP95, BGM00]). All these examples make use of the different derived constructs, thus programs become much more intuitive.

A very simple example of a tccp program which does not use derived constructs can be seen in Figure 5.3. This program is a counter. It calculates a series of numbers which start by Init.

```
counter_tccp(Init,I) ::= ∃x(tell(I = [Init | X])
                          || (∃Y(Y is Init + 1)
                          || counter_tccp(Y,X)))
```

Figure 5.3: tccp example: a counter

Let us now consider a bit more elaborated example. In Figure 5.4 the reader can observe a figure which represents the behaviour of a microwave. We can see that, for example, if we are in a state where it is satisfied that the door of the microwave is closed, the system is turned-off and no error has been detected, then if we open the door, we move to the state on the top of the figure.

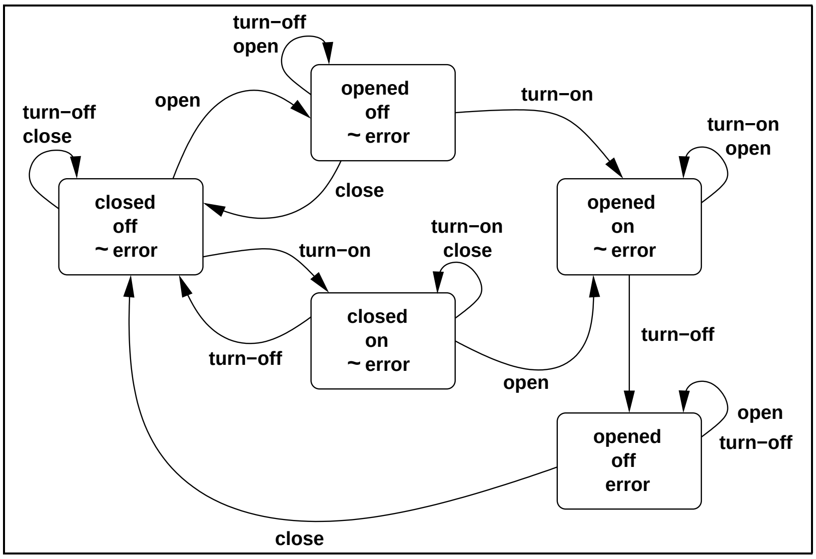

Figure 5.4: Example: the microwave system.

Now we can show the tccp program which models a part of the microwave model. In particular, the program showed in Figure 5.5 models the part which detects if the door is opened when the microwave is turned-on.

The whole system example is inspired in the system for a microwave control showed in the classical literature. However, for the tccp program we have considered only a subpart of the whole system in order to easily use it as a reference example in the following chapters of this thesis.

```
microwave_error(Door, Button, Error) ::=
        ∃ D, B, E(
              (now (Door = [open | D] ∧ Button = [on | B]) then
                     (∃ E1(tell(E = [yes | E1])) ||
                     ∃ B1(tell(B = [off | B1])))
              else
                     ∃ E1(tell(E = [no | E1]))) ||
              microwave_error(D, B, E)).
```

Figure 5.5: Example of a tccp program: a simple error controller

The reader can observe that the program in Figure 5.5 checks with a Conditional agent if the door is opened when the microwave is turned-on. In that case, it forces with the Tell agents that in the following time instant, the microwave is turned-off and an error signal is emitted. If it is not true that the door is opened and the microwave is working on, then the program simply emmits (Tell agent) a signal of *no* error in the following time instant. Therefore, this example corresponds to the part of the whole system which avoid wrong behaviors (in Figure 5.4 it is represented by the two states on the right).

5.4 tccp vs tcc

The two languages presented in Chapter 3 and Chapter 5 comes from the same theory: the cc model. Both the languages have been designed for modelling reactive systems and make the assumption of the *bounded asynchrony* hypothesis (see [SJG94a]). The bounded asynchrony hypothesis ensures that computations are not instantaneous but take a bounded amount of time. However, they have some important differences.

Let us discuss the main differences between tcc and tccp. The first interesting question that we must remark is the fact that, although both languages come from the cc paradigm, tcc is inspired by the synchronous languages approach while tccp is inspired by the process algebras approach. Therefore, they are essentially different and this motivates the following more concrete differences.

In this chapter we have seen that the tccp language is non-deterministic. However, we know that tcc is a deterministic language since it comes from the deterministic fragment of cc. This makes tcc more appropriate to model small embedded systems such as controllers whereas tccp is able to model more complicated systems.

Another divergence is the fact that the tcc language considers a notion of interleaving for concurrent processes while tccp makes the assumption of infinite processors and uses the notion of maximal parallelism.

The two languages are designed for modelling different kind of systems. The language defined by Saraswat *et al.* was defined for programming real-time kernels, while the language introduced by F. de Boer *et al.* provides a formalism for specifying large concurrent timed systems where non-determinism is crucial.

In Figure 5.6 we show the hierarchy of the cc languages. We can see in the top of the figure the cc paradigm. Arrows show the different evolutions or extensions of the cc model.

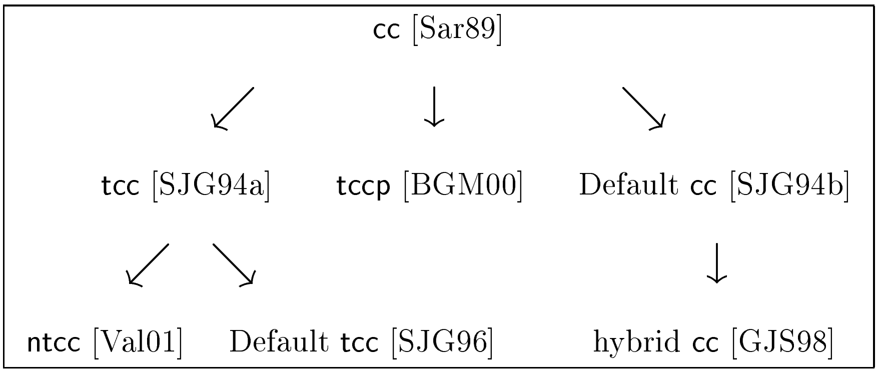

Figure 5.6: Hierarchy of the cc languages

As the reader can observe, in the last years many different extensions over time have been presented in the literature. There are approaches which extend the cc paradigm with a notion of *discrete* time and there is also an extension of the model with a notion of *continuous* notion of time (or *dense* time). The language which uses a notion of continuous time is called the *hybrid* cc language, which is able to model *hybrid systems*. We will present this language in Chapter 12. The other languages that appear in the figure are languages with a notion of discrete time.

Default languages introduce the notion of default into the language in order to handle permanent or default information. These languages allows to model strong preemptions when it is necessary to describe more precisely the behavior of systems.

ntcc is an evolution of the tcc language presented in this thesis. Actually it can be seen as an extension of tcc over non-determinism, thus it augments the expressiveness of the language. [Val02] gives more details about this language.

6

Verification Techniques

Verification techniques can be applied to both hardware and software systems. In this thesis we focus our attention on software systems and thus programs. Program verification consists in formally proving that a program satisfies some logical specification. If we consider as starting point the [Flo67] and [Hoa69] works, verification seemed a very promising approach to get correct software, but after 30 years of research, there is no general technology which can be used widely. There are two main reasons for this:

Dimension of specifications. The idea underlying the verification process is to compare two descriptions of the system behavior: the program itself and a (possibly) shorter and more abstract logical specification. The problem comes when we try to verify a complex system. In such cases, logical specifications can be too big, thus it is difficult to define them in a correct way. Note that since logical specifications are a reference during the verification process, it is crucial that they are correct.

Size of the proofs. The verification process needs very large proofs. It is easy to make errors during the proofs since they are tedious and boring. Thus it is difficult to check them manually in a reliable way.

The ideal situation would be to have a *fully automatic* verification technique. Such a verification tool or algorithm would take as input both the system and the logical specification (that such system must satisfy) and decide if the program satisfies the logical specification or not *without* the interaction of the user.

Computer science theory tells us that such algorithms cannot be defined in an effective way for a generic class of systems but only for small subclasses (see [Sip96, LP97, HU79, Pap94] for theoretic results). However, although this natural limitation is always present, practical solutions have been found in the last years. The underlying ideas are:

- it is possible to use abstraction techniques to hide or remove some details of the system that we want to verify. In this way, we can obtain an *abstract model* of the system which is simpler and can be verified in place of the original system,

- we can also restrict the class of systems to be verified. For example, all classical approaches to the model checking technique restrict the application domain to finite-state systems,

- it is also possible to restrict the verification problem to the essential parts of a program . For example, we could check only the communication protocols of a concurrent system in place of the whole system. In this way we could say that the most important part of the system is correct,

- we can combine automatic and non-automatic (or semi-automatic) verification methods. For example, we can use the deductive reasoning (non automatic) technique to verify that an abstract model preserves a specific set of properties of the system. Then we can apply an automatic technique to verify the abstract model.

In the following sections we describe the main features of the verification methods first studied in the literature. The first one is *theorem proving*, a deductive reasoning method principally guided by the user. The second method is *testing*, which is based on the analysis of *some* executions of the system. Then we introduce more in detail the approach which is used in this thesis: the *model checking technique.*

6.1 Theorem Proving

Usually, logic is considered as a framework for the deductive reasoning where, given a set of formulas that describes the domain and by applying to this set some deductive rules, the formula that must be checked is deduced. Therefore, the idea is to translate the program into formulas and to axiomatize the set of objects used by the program. Then we have to make the deductive proof of the program formula.

Theorem proving was the first technique to be defined for formal verification. The idea was introduced by Floyd and Hoare in [Flo67, Hoa69]. The verification process used by this technique is performed essentially manually, thus it can be very difficult and errorprone. In order to solve these problems, powerful tools have been developed to make the process semi-automatic. *Theorem provers* can help the user to make correctness proofs. Some of the most popular theorem provers are Isabelle [NPW02], PVS [ORS92] or sTeP [MAB+94].

Many of these tools, by using some heuristics, can suggest the user how to continue a proof at a specific point. Furthermore, theorem provers provide a very powerful mechanism to detect and handle incomplete proofs or even to save proofs to be used later in other proofs.

Theorem proving has many good features. For example, it is a very reliable technique because it uses mathematics and logic theory in a very rigorous way. Moreover, this technique is not limited to finite state systems. Theorem proving allows one to verify programs which use lots of domains and data structures. It is even possible to verify parametric programs (for example, programs which have a finite but not specified number of identical processes).

Usually, verification is not made by the programmer but by a different person, thus it is easier to find errors than in other techniques where both the tasks are performed by the same person. Another good thing is the fact that *invariants* are introduced in the code. An invariant is a condition over system variables or program counters that must be always satisfied during execution. Thus, invariants also allow us to perform run-time verifications.

Finally, theorem provers can be helpful in the definition of formal semantics of programming languages. Proofs can ease the understanding of the program that has to be verified. For example, if a construct is difficult to integrate in the proof system, then it could be because the construct is not well defined or because we have not correctly understood its meaning.

The theorem proving approach has also many drawbacks. One of the most important is the fact that a lot of time and efforts are needed to complete a proof. We could say that this is the most inefficient formal verification technique. Often, it is necessary more time to verify a program using the deductive verification approach than to code the program itself. Moreover, the success of the verification process depends a lot on the intuition and capacity of the person who makes the proof: this is a technique that must be used by people expert in mathematics and logic. The most difficult part of the proofs is the introduction of invariants and assertions. Some heuristics have been defined in order to make this task automatic, but in general one has better results with hand introduced information.

Automatization can also avoid another common error. The user can use shortcuts in order to finish the proof earlier. Therefore, pieces of code which seem trivial to the user could be not proved. Some automatic theorem provers avoid this risk by requiring the total verification. Another problem is that the user could introduce too many assumptions. This fact makes the proof less general since it is restricted to particular cases. In those cases what happens is that people adds assumptions as axioms into the proof by guessing that they are obvious and make the proof shorter.

In conclussion, although theorem proving is very reliable due to the formalism, many times the verified system is not the program code itself but some abstraction of it. Therefore, we cannot immediately ensure the correctness of the verification but it is also necessary to verify the correctness of the abstraction or simplification.

6.2 Testing

Testing (see [Mye79, Bei90, KFN93, Pat00]) is a verification method which consists in executing the program that we want to verify and analyzing the executions (patterns) to detect errors. Each pattern is compared with the expected result and if they do not coincide, then an error is signed.

This technique is based on the analysis of only *some* of the possible executions of the system, thus the goal is not to prove the total absence of errors in a program. For this reason this technique cannot be considered a *formal method*. However, testing is widely used to improve the quality of software.

There are many kind of proofs which are applied in different levels and phases of the verification process. For example, *unit proofs* are the lowest level proofs. These proofs are applied to small pieces of code independently. *Integration proofs* check the different pieces of code to be well integrated. In a higher level, *system proofs* verify the correctness of the program in general (the global output) and finally, *acceptance proofs* are made by the final user and determine if the system satisfies all its requirements.

Intuitively, low level proofs determine where an error occurs in the code, whereas high level proofs give us an approximation of the cause of the error. Moreover, low level proofs allow us to verify concurrently different independent pieces of code.

We can also distinguish two possible ways to do the verification. On one side we have *white box* tests (see [Col00]) which are based on the analysis of the internal details of systems. On the other, *black box* tests (see [Bei95]) do not consider the internal code of the system but only its input and output data. Usually, black box tests are used for the high level proofs (acceptance proofs for example).

As we have said before, testing is based on the fact that the number of execution paths of a program can be very large, even infinite, thus it is impossible to analyze *all* the *execution paths* in an efficient way. An execution path is a sequence of control points and instructions appearing in the program source code.

The set of execution paths which are analyzed are selected by using some heuristics or criterium. Such criterium must provide a high probability to find errors. For example, we can choose the set of paths for which each sentence of the program code appears in some selected execution path. Another criterium can be to require that each arc of the flow diagram of the system appears in some analyzed path. The quality of the testing study (*text suite*) can be measured depending on the coverage of the execution paths over the program.

In the black box approach, the system is usually modelled as a graph or an automaton, then graph algorithms are applied to the model in order to generate proof cases which will be executed in the system. In this case, the different heuristics or criterium of coverage are applied to the graph or automaton. In the white box approach the case studies depend on the source code.

There are different classes of tools which help the user to make the proofs. Some of them are used to generate proof cases, evaluate the coverage of the set of selected paths, or to execute proof cases. Some of the most popular testing tools used in the literature are GCT (Generic Coverage Tool), PET (Path Exploration Tool, see [GP99]), TestMaster, TestPartner, TestWorks/Coverage, CodeText or TestCenter.

Testing has many good characteristics. For example, this technique is applied directly to the source code, thus it is not necessary to construct a model of the system as occurs in theorem proving. This technique is widely used to improve the quality of software since it is a very simple technique. It is not necessary to use complicate mathematics or logic formalisms, thus any user can use it.

Testing gives a practical solution for the verification problem for large (even infinite-state) systems with a reasonable cost. It needs less time and resources than the deductive verification approach. Furthermore, this is a modular technique where different pieces of code can be verified in an independent way. This fact allows the verification to be (partially) performed before the end of the programming phase of the whole system.

Unfortunately, to use testing have also some disadvantages. The main drawback coincides with the reason why it cannot be considered a formal verification technique: testing is not an *exhaustive* technique, thus we cannot ensure the total absence of errors in the system. In general, testing is less reliable than other verification methods such as theorem proving.

Often, the source code must be manually modified in order to calculate statistics between some variable and its values, or to make comparisons between variables. Therefore, although the method can be to the source code, actually the original code must be modified.

Finally, proof cases are chosen depending on different heuristics. Thus, the testing results depend on the used heuristic. Moreover, this technique is usually applied by the programmer of the system. The programmer could trust too much in his program making the verification less severe.

6.3 Model Checking

The third formal verification technique that we consider is the *model checking* technique. Model checking was first introduced by Clarke and Emerson [CE81, EC80] and by Quielle and Sifakis [QS82] independently. This technique has been studied deeply in the last two decades and has become a very important research line. The reader can find a wide overview of model checking in [CGP99, BBF$^+$01].

Model checking was defined as an *automatic* verification technique (see [CES86]). The method is based in a quite simple formal logic problem: to check if a specific property (expressed as a temporal logic formula) is satisfied in a specific finite domain (a transition system which represents the system). In other words, model checking try to verify if any execution of a transition system is a *model* of the formula. Here, by a model of a formula is intended a sequence of states which satisfies a given specification. We have to remark that usually, in the model checking literature, the word *model* states for the transition system which represents the system that must be verified.

In fact, in order to check if a program satisfies a specific property it is sufficient to check the corresponding temporal formula over the transition system. States of the transition systems represent all the possible states of the program whereas transitions from one state to another represent the execution of a program instruction.

Transition systems used in model checking are generic models of the system. These models usually are non-deterministic. The non-determinism of the transition system comes from the concurrency (which is modelled by interleaving) or from the absence of information about the behavior of some component of the system or its environment. For example, an arbitrary value provided by the environment can be modelled by a non-deterministic choice of such value.

Concurrent systems can be very complicated, thus model and verify them manually can be too hard. The development of formal and (fully) automatic verification methods such as model checking is essential. Basically, model checking consists in an exhaustive analysis of the state-space of the system. This exhaustive analysis implies that we can apply it *only* to finite state systems.

Therefore, model checking does not intend to be a general method. It was thought to be applied, for example, to systems that have a short description for states; systems for which control is more important than data as hardware, or concurrent protocols, process control systems or, more generally, *reactive systems* (see [Pnu86]).

Reactive systems are defined as those system which interacts along the time with their environment. Many times these systems do not terminate but runs forever, thus a reactive systems cannot be modelled by its input-output behavior since there is no a *final result.*

It is also possible to apply this verification technique to a more complex class of systems: hybrid systems. These systems evolve continuously over time but are controlled by a discrete component. The reader can find more details about hybrid systems and its verification in [HMP92, HHWT95].

Classical model checking can be divided into three main tasks. The first task consists in converting the design of the system into a formalism that can be handled by a model checker. In the classical literature state transition graphs are used for such first task. The second task is to specify those properties that the system must satisfy. For this purpose classical logical formalism such as temporal logics are used. Finally, the third task is the verification itself. Theoretically it is an automatic process, but in practice some human assistance is needed in order to interpret the verification results. If we obtain a negative result, the model checker provides a counterexample. The user can analyze the error trace and identify where the error is localized.

6.3.1 The state explosion problem

The main problem of the model checking technique is that the *state-space* of a concurrent system can be huge. For example, a system with n identical processes each of them having m states could have m^n states. The number of states limits the applicability of the technique since transition systems become too big (in some cases infinite) and it is difficult (or impossible) to build them. This question is called in the literature *the state explosion problem.*

Many researchers had tried to mitigate the state explosion problem but there is no a generic solution for the problem. Here we introduce some of the different strategies presented in the literature.

In particular, the different approaches can be classified in two categories: (1) those techniques which try to build only the part of the state-space needed to check the property, and (2) those symbolic approaches that represent symbolically the states instead of enumerate them explicitly.

Some of the approaches in the first category are the following:

- *partial order reduction* techniques ([God90, Val90, Pel93]) try to avoid the representation of all the possible sequences of states. The key idea is the commutativity between concurrent transitions when they are equivalent under a specific property,

- *on-the-fly* approaches ([JJ91, CVWY92, Cou99]) build only the section of the state-space which is needed to check the temporal formula. Here we can say that the construction of the state-space is guided by the property.

- *symmetry* ([ES93]) takes advantage of the permutations on the components of a state which provoke the same executions for a specific property. The approach is based on the fact that some executions cannot be distinguished when we interchange some of its concurrent processes. A unique representant of each equivalent class (under permutation) is used.

Abstraction can be introduced in both the categories. If the method verifies a simpler model which is obtained by removing some irrelevant details depending on the property, then we can say that abstraction is a technique of the first category. However, if the classical model is constructed and then, an abstraction is applied to such model in order to reduce the size of the transition system, then we classify such method into the second category.

The second line of research that tries to solve the state explosion problem is the *symbolic model checking* approach ([McM93, BCM$^+$92]). The idea of this technique is to represent implicitly the states and transitions of the transition system which models the program. Usually, the implicit representation used in literature are the *Binary Decision Diagrams* (BDDs) [Bry92, Bry].

In his PhD thesis ([McM93]), McMillan introduced the symbolic model checking. The symbolic representation was based in the *Ordered Binary Decision Diagrams* (OBDDs) defined in [Bry]. OBDDs are essentially an efficient codification of boolean formulas. The key idea is that temporal formulas can be checked directly over the implicit representation of the system. This approach uses the branching temporal logic CTL (*Computation Tree Logic* [BAMP83]) to specify the property.

Symbolic model checking allows us to handle very complex systems. In particular, the initial algorithms were able to verify systems with more than 10^{20} states.

Following refinements allowed scientists to verify systems up to 10^{120} states. However, the problem remains in general the same, and still infinite-state systems cannot be verified.

6.3.2 Complexity

There are two major families of temporal logics used in the model checking literature: the *branching time temporal logic* and the *linear time temporal logic*.

In the branching time temporal logic approach (see [CE81, CES86]), the temporal operators of the logic consider *all* the possible successors or *any* of them. For example, it is possible to say that "in some future state" a property will be satisfied, or that "in all the reachable states" a property will be true.

The linear time temporal logic approach (see [LP85, VW86]) was defined to describe linear sequences of states, thus each state has only one successor. The problem of model checking for linear time logic consists in verifying that *all* the linear sequences that can be generated by the transition system satisfy the linear temporal logic formula.

The model checking algorithm for the branching-time logic CTL presented in [CES86] is linear in the product of the length of the formula and the size of the state transition graph. Thus, first model checking algorithms were able to analyze systems with 10^4 or 10^5 states. On another way, model checking for CTL* or LTL is a more complex problem since they are PSPACE-complete problems.

This last result could be surprising for the reader because CTL* is a more expressive logic than LTL but it can be explained by the fact that in both cases model checking algorithms must construct all the models of the systems in order to verify the property.

Finally, we can say that the linear approach is the most appropriate when we want to verify properties of executions of the program. However, if we want to verify properties of the structure of the program, then the branching approach is the more suitable. In [Wol87] and [Eme90] the reader can find more details in the adequacy of each approach.

6.3.3 Model checking characteristics

Model checking has many good features. The most important one is the fact that it is an *automatic* verification technique. Usually, the model must be constructed manually, thus in general we cannot say that the process is fully automatic. Nevertheless, the user must make quite less manual work than using other verification methods.

Another good thing of model checking is that it is possible to combine abstract interpretation with the model checking technique. Note that the complexity of the automatic verification of software is huge, thus abstractions could reduce the cost of the verification process.

Furthermore, it is *quite easy* to implement tools. Model checking is based on simple ideas, thus it is easy to define algorithms which implement them. There have been defined many software tools (*model checkers*) which allow us to verify many properties. These model checker are developed *ad-hoc* by defining optimized algorithms for each particular class of systems. Therefore, it is very important to chose the right tool for the verification of a specific property and program. Some model checkers used in the formal verification community are Spin [Hol91, Hol97], SMV [CMCHG96], VIS [Vg96], COSPAN/FormalCheck [HK90], Murphi [DDHY92] or PEP [Gra97, Gra99].

Usually, model checking algorithms provide a *counterexample* in the case that the formula is not satisfied by the system. The counterexample consists of an execution of the system which does not satisfy, and it allows the user to debug the original system. Often it is more interesting a counterexample provided by the model checking tool than the verification of the property itself.

Finally, it is important to remark that it is easy to use a model checker. Moreover, it can be used repeated times over the same program in order to check different properties obtaining very useful information about the system. Model checking can be viewed as a “detailed” debugging tool.

Although the state explosion problem is the main problem of the model checking technique, there are other drawbacks. The first one could be the fact that the system *must be* (manually) modelled in the language handled by the model checker. Sometimes there exists an automatic parser from the source code to the syntax used by the model checker. However, some optimizations and heuristics integrated in some tools can be used only when the user models the system by himself. He could also introduce some abstractions in the system.

Moreover, as for the deductive verification approach, it is usually verified a simplification of the program in place of the original system. Thus, the output of the model checker must be carefully analyzed since the model can be an over-approximation or under-approximation of the system and the result could be dishonest. Counterexamples are very important since they allow the user to check the correctness of the result.

In some cases the user must provide some parameters for the configuration of the model checking tool. Therefore, the user must have some knowledge about the tool and algorithms in order to use them. Termination of the verification process is not guaranteed by all the tools in the literature. In addition, it is difficult to know a priori if the tool is able to check a specific property for a given system. This is in part due to the high variation on the state-space of a single system when verifying different properties.

6.4 Infinite state model checking

As we have said before, model checking was introduced to verify finite-state systems, but there is a special interest in the application of this technique to infinite-state systems. The reader can find very promising approaches to deal with the state explosion and infinite-state problems in the literature.

For example, an important research line proposes the use of *abstract interpretation* ([Dam96, CC77, CC92]) in order to obtain an abstract version of the system which is finite and computationally treatable; another approach, that is mostly applied to timed automata, calculates a quotient of the infinite set of states obtaining an equivalent automata with a finite number of states (see [AD90] for details).

The infinite-state approaches can also be classified into two categories. Firstly, we have those approaches that work *symbolically* by exploring the infinite state-space. Some of these approaches define abstractions to construct a finite model of the system which can be automatically verified (see [CGL94, LGS$^+$95]).

Secondly, there exists approaches which are based on the symbolic reachability analysis. Intuitively, these approaches reduce the state-space when they analyze the model of the system. Essentially, a finite representation of the set of all reachable configurations of the system is calculated (see [ACH$^+$95, CH78, BEM97, BG96]). It is defined a finite quotient of the large (infinite) set of states of the systems obtaining a (equivalent) finite state system.

The first kind of approaches can be viewed as *representation* approaches while the second ones can be viewed as *reduction* approaches. Intuitively, representation approaches try to represent the system behavior using a compact formalism. Reduction approaches take the model of the system and try to analyze only the subset of states which is necessary for the verification of the property. Many times the subset of states that must be analyzed is determined by the property and the system itself. The reader can observe that this classification is in someway similar to the one presented in Section 6.3.1.

For example, the quotientation technique used for *timed automata* can be classified as a reduction technique while the abstract interpretation can be seen as a representation or a reduction technique since it can be used to construct an abstract model of the system (representation approach) or we can apply abstract interpretation to the (classic) model of the system (reduction approach). The final objective of all these approaches is to reduce the large state-space to a finite number of states.

There are other methods which do not follow this classification. The approaches that use regular languages and regular relations are considered in the so called *regular model checking* method ([PS00, KMM$^+$97, BJNT00]). In [AAB$^+$99] the reader can find an approach which consists on combining the notion of abstraction and the notion of symbolic reachability in order to define a method to verify infinite state systems.

6.5 Model checking for the cc paradigm

The cc paradigm has some interesting features which allow us to define a model checking algorithm for reactive and hybrid systems. We consider two languages which have a notion of time in their semantics. We use a time interval (provided by the user) in order to restrict the state-space of the system. The fact that the time is in the semantics makes reasonable the use of such restriction since the user knows how much time is needed to have a response from the system.

The reader could think that the restriction to an interval of time could make the algorithm incomplete in many cases. Note that it does not occur since many times such limit of time is not reached (obviously the user must provide a reasonable time interval). Moreover, if the limit is reached we obtain an over-approximation of the system thus some properties can still be checked. The idea to limit the verification to a limit of time is not new. It has been used in different approaches, for example in [AHWT97]. Also interval logics that play a similar role are widely used in the literature (the reader can find an example in [KRM$^+$93]).

The tccp model checking problem consists in applying the model checking technique to tccp programs. The idea is to use the notion of constraint which is underlying the language in order to have a compact model of the system first, and second, to handle the model directly to verify properties.

First of all, as we have said, we use constraints in the *automatic* construction of the model. Constraints represent in a compact way a set of possible values that a variable can take (i.e., a set of states if we use the classical notion of state). Then, in the second phase we take advantage of constraints for specifying the property which we want to verify by using a logic that is defined to handle constraints. Such logic is presented in [BGM01] and is able to work over constraints in a very intuitive way.

The last phase of the model checking technique consists in defining an algorithm that determines if the system satisfies the property. We use the two outputs of the previous phases to adapt the classical algorithm defined for LTL to our framework. We essentially follow the same steps of the original algorithm. Note that this is possible because we use a logic that works over constraints, thus allows us to verify properties directly over the tccp Structure defined. If we use a classical temporal logic, we must transform the tccp Structure into a Kripke Structure over which temporal formulas can be checked.

The problem for hcc is similar to the problem for tccp. The idea is the same, i.e., to define a structure able to represent the system behavior and to check properties over such structure. However, as a first step we have constructed the model and we have transformed such model into a linear time automaton which can be given as input to a classical model checker such as HyTech.

Therefore, although we can model any hybrid system using our hcc Structure, in this thesis we have limited ourselves to linear hybrid systems since HyTech is able to handle only such subclass of hybrid systems.

Note that in our approach we automatize the construction of the model while in almost classical approaches it is constructed by hand.

8

A model for tccp programs

In this chapter we describe in detail the first phase of our model checking approach. The idea is to automatize the construction of the model of the system from the specification. In other words, we take a program written in tccp, and a model of the system behavior is constructed in an automatic way. As we have said before, in [FPV00a, FPV00b] it was presented a method to construct a structure as a first step towards the definition of a model checking technique for tcc. Nevertheless, the structure defined in [FPV00a, FPV00b] to model tcc programs was quite different from the structure defined in this work. Let us now introduce the necessary notions for the construction of our model.

8.1 Labelling

First of all, we need a labelled version of the specification in order to construct the model of the system automatically. We adapt the idea introduced by Manna and Pnueli in [MP95] to our framework: a different label is assigned to each occurrence of an agent. Labels allow us to identify in which point of the execution of the program we are. The presence or absence of a label determines if the associated agent can be executed or not during the computation.

The labelling process consists on the introduction of a different label for each occurrence of a language construct. Thus, we take a tccp program and modify it by adding labels:

Definition 8.1.1 *Let* P *be a specification, the labelled version* P_l *of* P *is defined as follows:*

- *If* $\mathsf{P} = \mathsf{abort}$ *then* $\mathsf{P}_l = l_{\mathsf{abort}}\,\mathsf{abort}$.
- *If* $\mathsf{P} = \mathsf{tell}(c)$ *then* $\mathsf{P}_l = l_{\mathsf{tell}}\mathsf{tell}(c)$.
- *If* $\mathsf{P} = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i$ *then* $\mathsf{P}_l = l_{\mathsf{ask}} \sum_{i=1}^{n} l_{\mathsf{ask}(c_i)}\mathsf{ask}(c_i) \to l_{A_i}$.
- *If* $\mathsf{P} = \mathsf{now}\ c\ \mathsf{then}\ A\,\mathsf{else}\ B$ *then* $\mathsf{P}_l = l_{\mathsf{now}}\,\mathsf{now}\ c\ \mathsf{then}\ A_l\ \mathsf{else}\ B_l$.
- *If* $\mathsf{P} = A \parallel B$ *then* $\mathsf{P}_l = l_{\parallel}(A_l \parallel B_l)$.

- *If* $\mathsf{P} = \exists x\, A$ *then* $\mathsf{P}_l = l_e \exists x\, A_l$.
- *If* $\mathsf{P} = \mathsf{p}(x)$ *then* $\mathsf{P}_l = l_p \mathsf{p}(x)$.

Thus, we explore the tccp specification and each time that we find an occurrence of a construct we introduce a new label which identify such point of the program. There will be one different label for each occurrence of an agent.

In Figure 8.1 we show the labelled version of the microwave errpr detection program example showed in Figure 5.5. Note that the structure of the program has not changed, simply some labels have been added.

```
$\{l_0\}$ microwave_error(Door, Button, Error) ::=
        $\{l_{e_1}\} \exists$ D, B, E(
            $\{l_{\|_1}\}$ ($\{l_c\}$ now (Door = [open | D] $\wedge$ Button = [on | B]) then
                    $\{l_{\|_2}\}$ ($\{l_{e_2}\} \exists$ E1($\{l_{t_1}\}$tell(E = [yes | E1])) ||
                    $\{l_{e_3}\} \exists$ B1($\{l_{t_2}\}$tell(B = [off | B1])))
            else
                    $\{l_{e_4}\} \exists$ E1($\{l_{t_3}\}$tell(E = [no | E1]))) ||
            $\{l_p\}$ microwave_error(D, B, E)).
```

Figure 8.1: Example of a labelled tccp program: a simple error controller

8.2 The tccp Structure

The main point of the modelling phase is to construct the structure which models the system behavior. We define a new graph structure to represent the system. The tccp *Structure* is defined as a variant of the *Kripke Structure*. The main difference between the two structures is that the definition of a state in the Kripke Structure follows the classical notion of state (a state is defined as a valuation of the set of variables of the system) whereas in our structure, a state consists of a conjunction of constraints and can be seen as a set of classical states.

First of all we remark the similarities between the classical graph structure used in the literature (for example the Kripke Structure[1] defined in [CGP99]) and the structure defined for this approach.

Intuitively, a Kripke Structure is a *finite* graph structure where there could be many initial nodes and each node is always related to another one (or to itself). Moreover, each state has associated a set of atomic propositions which are true in such state.

We now define our graph structure: the tccp Structure, which allows us to model the behavior of systems specified in tccp. First of all we define what the set AP of atomic propositions is:

[1] See Chapter 1 for the formal definition.

Definition 8.2.1 *The set* AP *of atomic propositions is defined as the set of elements*[2] *of the Cylindric Constraint System of the* tccp *language considered*[3]*.*

In the rest of the paper we abuse of notation by identifying the meaning of the terms *constraint*, *atomic proposition* and *element*. Now we define what a state of the tccp Structure is:

Definition 8.2.2 (tccp State) *Let* AP *be the atomic propositions in the* tccp *syntax and* L *be the set of all possible labels generated to label the original specification of the system. We define the set of states as* $\mathsf{S} \subseteq 2^{\mathsf{AP}} \times 2^{\mathsf{L}}$

Finally, we define the tccp Structure. Observe that it is very similar to a Kripke Structure. Actually, the only differences are the definition of state (in Definition 8.2.2) and the two labelling functions C and T which replace the labelling function L_{KS} of Definition 1.3.1.

Definition 8.2.3 (tccp Structure) *Let* AP *be a set of atomic propositions, we define a* tccp *Structure* M *over* AP *as a five tuple* $\mathsf{M} = (\mathsf{S}, \mathsf{S}_0, \mathsf{R}, \mathsf{C}, \mathsf{T})$ *where*

1. S *is a finite set of states.*
2. $\mathsf{S}_0 \subseteq \mathsf{S}$ *is the set of initial states.*
3. $\mathsf{R} \subseteq \mathsf{S} \times \mathsf{S}$ *is a transition relation.*
4. $\mathsf{C} : \mathsf{S} \to 2^{\mathsf{AP}}$ *is the function that returns the set of atomic propositions in a given state.*
5. $\mathsf{T} : \mathsf{S} \to 2^{\mathsf{L}}$ *as the function that returns the set of labels in a given state.*

We assume that a transition in the graph represents an increment of one time-unit in the system. Intuitively, C labels a state with the set of constraints true in such state. In other words, this function represents the information that we know in a specific instant (i.e., the store of the system). T labels each state with the set of labels associated to agents that must be executed in the following time instant. In other words, T represents the point of execution in each instant (or state).

When two states s and s' are related by relation $\mathsf{R}(s, s')$, it means that it is possible to reach the state s' from the state s by executing the agents associated to the labels in $\mathsf{T}(s)$ with the store $\mathsf{C}(s)$ deriving as a result the store $\mathsf{C}(s')$ and the point of execution $\mathsf{T}(s')$.

Given a tccp Structure T, we define $tr(\mathsf{T})$ as the set of sequences of states of T starting from an initial state and which are related following the order in the sequence:

$$tr = \{s \mid s = s_0 \cdot s_1 \cdots s_n \cdots \wedge s_0 \in \mathsf{S}_0 \wedge \forall i \geq 0, \exists \mathsf{R}(s_i, s_{i+1})\} \qquad (8.2.1)$$

[2] See Definition 1.4.5.

[3] Note that tccp is parametric w.r.t. a Cylindric Constraint System, see Chapter 5 for details.

8.3 Construction of the model

Now we explain how the tccp Structure is constructed from a labelled specification S in an automatic way. This is the main point of the construction of the model of the system. The resulting graph will represent the behavior of the system.

The main procedure is showed in Figure 8.2. Given a tccp declaration D of the form $p(x) :: l_A A$, the function construct(D) returns a tccp Structure $Q = \langle S, S_0, R, C, T \rangle$ representing the behavior of p. We simplify the treatment of functions C and T. Although we do not mention them in the algorithm itself, these functions corresponds to the two components of the state structure of the algorithm. We also write $R(n, n')$ to describe that nodes n and n' are related.

In the algorithm showed in Figure 8.2 the tccp Structure is initialized and the set of initial states is created. Then the function construct_ag (in Figure 8.3) is called. This functions iteratively completes the construction. Functions instant() and follows() are defined below. The $\aleph$ value is a possible value of the elements in *inf*.

```
construct(input D : tccp declaration, output ⟨S, S_0, R, C, T⟩ : tccp Structure)
    state :
        st : store;
        ℓ : set_label;
    n[] : state;
    inf[] : store;
    lab[] : set_label;
    m, j : int;

    inf = instant(true, $l_A$);
    lab = follows($l_A$);
    m = sizeof(inf);
    for j = 1 to m
        if inf[j] <> $\aleph$ then
            n[j] = create_node(inf[j], lab[j]);
            $S' = S' \cup$ n[j];
            $S'_0 = S'$;
    construct_ag($S', S'_0, R', C', T', n$);
    $S = S'$; $S_0 = S'_0$; $R = R'$;
    $C = C'$; $T' = T$;
```

Figure 8.2: Description of the construction algorithm

Intuitively, the construction evolves as follows. A process is composed by a set of clauses and a goal. A *specification* is the set of clauses of a process. In this section of the chapter we describe how a specification (or declaration) can be transformed in a set of tccp Structures. Actually, for each different clause we construct a tccp

Structure which is labelled with a unique name. This name can be used as one of the labels introduced in the labelling phase when a procedure call agent is analyzed.

We also define $\mathsf{follows}(ll)$ and $\mathsf{instant}(c, ll)$ which are functions used during the construction of the tccp Structure. We consider that the declaration D of the form $p(x) :: l_A A$ is an always available public information.

Given a label ll, $\mathsf{follows}(ll)$ returns the list which contains the labels associated to the agents that must be analyzed in the following time instant. Each element of the list corresponds to a different possible behavior of the system. For example, if there is a conditional agent, the inital part of the list corresponds to the possible behaviors when the guard of the agent is satisfied and the final part of the list corresponds to the case when it is not satisfied. Therefore, if two or more conditional agents are nested, then all the possible behaviors depending on the first *then* part will appear before than those of the *else* part in the list. Since tccp restricts the number of nested conditional agents in a program, we can ensure that this algorithm terminates and the list of sets of labels is finite.

The follows algorithm uses two addicional auxiliar functions. append and combine are functions that implement operations over lists. In particular, $\mathsf{append}(\ell_1, \ell_2)$ returns the concatenation of the two lists ℓ_1 and ℓ_2. $\mathsf{combine}(\ell_1, \ell_2)$ constructs a new list whose elements consists of an element of ℓ_1 and an element of ℓ_2. For example, if $\ell_1 = \{\{l_1\}, \{l_2\}\}$ and $\ell_2 = \{\{l_3\}\}$, then the result of $\mathsf{combine}(\ell_1, \ell_2)$ is the list $\{\{l_1, l_3\}, \{l_2, l_3\}\}$.

Note that we always assume that each label l_A is associated with the agent A.

We can show that the complexity of the algorithm showed in Figure 8.4 is exponential in the maximum number of nested agents in the specification:

Lemma 8.3.1 *The time complexity for the algorithm* $\mathsf{follows}(A)$ *presented in Figure 8.4 is* $O(n * 2^m)$ *where* m *is the maximum number of nested agents and* n *is the size of the resulting list.*

Proof. First of all, we know that the agent A has a finite number of nested agents. Moreover, we can see that the cost of the algorithm in the case of *tell*, *abort*, *skip* agents is constant since $\mathsf{follows}(A) = \{\}$ in such cases. The cost is constant also in the case of *procedure call* agents since $\mathsf{follows}(p(x))$ returns a single label. For the *choice* agent, the cost depends on the number of asks contained in the agent. Therefore, given the agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$, the cost will be $n + 1$. In addition, we know that the maximum number of nested recursive calls is 2^m which corresponds with the worst case: when every nested agent is a parallel or conditional agent. Note that in that point, the functions combine or append are used. Assume that the cost of these functions is linear in the size of the resulting list. Then, we have the time complexity of the worst case $O(n * 2^m)$. Note that this is a theoretical case which does not occur in practice. We think that the complexity in practical cases should be semilinear on average. ■

We need also a second auxiliary function for the automatic construction of the model. Given a store and a label, instant(c, ll) returns the information which can be computed instantaneously (i.e., before the following time instant) by executing the agents associated with the label ll.

Note that we have marked the negation not(c) with a star to indicate that the semantics of negation is defined as the non satisfiability of c instead of the satisfiability of $\neg c$.

Now we present the auxiliary functions that are used in the previous algorithms. flat(st,ll) adds the constraint st to each element of the list ll returning a simple list of stores. If st is inconsistent with any element of the list, then the value of the element is setted to $\aleph$.

Note that the time complesity of flat is linear on the size of the list:

Lemma 8.3.2 *The time complexity for the algorithm flat(c,ll) presented in Figure 8.6 is* $O(n)$ *where* n *is the number of elements in the list* ll.

Proof. The proof is trivial since we iterate n times over the elements of the list. ■

Now we can show that the complexity of the algorithm instant showed in Figure 8.5 is exponential in the maximum number of nested agents in the specification.

Lemma 8.3.3 *The time complexity for the algorithm instant(c,A) presented in Figure 8.5 is* $O(n * 2^m)$ *where* m *is the maximum number of nested agents and* n *is the cardinality of the resulting list.*

Proof. We know that the agent A as a finite number of nested agents. We also know that if the agent is an *abort*, *skip*, *tell* or *procedure call* agent, then the cost of the function is constant. If A is a choice agent, then we have a linear cost, in particular we have $O(n + 1)$ since there is an iterative loop.

Now look to the three remaining cases. For both the *conditional* and the *parallel* agents we have two recursive calls, whereas for the *hiding* agent we have a single recursive call. We assume that the combine and append functions are linear on the size of the two lists passed as argument (i.e., we take $O(n)$ where n is the number of elements in the resulting list). Therefore, since we know that also the flat function is linear, we can say that the upperbound for the global complexity of the algorithm is $O(n * 2^m)$ where m is the maximum number of nested agents. Note that this is a theoretical case which does not occur in practice. We think that the complexity in practical cases should be semilinear on average. ■

Now we can analyze the complexity of the construct algorithm. First of all, we state the complexity for the construct_ag function. Then we define the global complexity of the algorithm.

Lemma 8.3.4 *The time complexity for the algorithm* construct_ag(S, S_0, n, R, C, T) *presented in Figure 8.3 is* $O(c * n * 2^m)$ *where* m *is the maximum number of nested agents and* c *is the number of states in the model and* n *is the number of elements in inf.*

Proof. By Lemma 8.3.3 and Lemma 8.3.1 we know the complexity of the auxiliary functions. Moreover, we know that select and remove takes linear time and we assume that create_node has constant complexity. We know that the while loop will be executed c times, where c is the number of different states in the model.

We can see that each time the while is executed, we have one procedure call to each auxiliary function. Moreover, we have a **for** loop which is executed n times where n is the size of the inf list. Therefore, the cost of the **for** loop is $O(n)$ and the cost of the **while** loop is $2c * (n * 2^m)$. We ensure the finiteness of the number of states since we know that there is a finite number of combinations of labels and constraints (which appear in the specification) modulo renaming. ■

Lemma 8.3.5 *The time complexity for the algorithm* construct(D) *presented in Figure 8.2 is* $O((c + 1) * (2n * 2^m)$ *where* m *is the maximum number of nested agents and* n *is the cardinality of the resulting list.*

Proof. We know the cost of the auxiliary algorithms. Following the structure of the algorithm, we can see that there is one call to the construct_ag function. In addition, we have a procedure call to the algorithm instant and follows. Then, we have to add the cost of such algorithms: $O(2n * 2^m + c * (2n * 2^m))$. We have also a for loop which is executed n times where n is the number of elements in inf. Therefore, we obtain the global complexity given in this result. ■

Formally, the same construction showed in the algorithms above can be described as a function ρ_d. This function, given a tccp Structure and a declaration, constructs the resulting tccp Structure by using the function ρ_a.

Definition 8.3.6 (Function ρ_d) *Let* $p(x) :: A$ *be a procedure declaration* D *where* A *is a tccp agent, and a tccp Structure* $Q = \langle S, S_0, R, C, T \rangle$*, we define the function* $\rho_d : D \times Q \times Q'$ *where* $Q' = \langle S', S_0, R', C', T' \rangle$ *is another tccp Structure as follows:*

$$S = S_0 = \{s_0\}, C(s_0) = \mathit{true}, T(s_0) = l_A$$

$$\mathrm{ref}_p = \rho_a(A, Q, s_0)$$

Definition 8.3.7 (Function ρ_a) *Let* A *be a tccp agent and* $Q = \langle S, S_0, R, C, T \rangle$ *a tccp Structure. Given* $s \in S$*, we define the function* $\rho_a : A \times T \times S \times T$ *where* $Q' = \langle S', S_0, R', C', T' \rangle$ *is another tccp Structure as follows:*

$\rho_a(A, Q, s) =$

```
    if $A \equiv \mathsf{abort}$ then
        $S' = S \cup \{s'\}, R' = R \cup \{R(s, s')\} \cup \{R(s', s')\}, C(s') = C(s), T(s') = \emptyset$
    if $A \equiv \mathsf{tell}(c)$ then
        $S' = S \cup \{s'\}, R' = R \cup \{R(s, s')\}, C(s') = C(s) \sqcup c, T(s') = T(s) \backslash \{l_{\mathrm{tell}}\}$
    if $A \equiv \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ then
        $S' = S \cup \{s'_i \mid c_i \textit{is consistent with } C(s)\} \cup \{s'\},$
        $R' = R \cup \{R(s, s'_i)\} \cup \{R(s, s')\}, C(s'_i) = C(s') = C(s),$
        $T(s'_i) = (T(s) \backslash \{l_{\mathrm{choice}}) \cup l_{A_i}, T(s') = T(s)$
    if $A \equiv \mathsf{now}\ c\ \mathsf{then}\ B\ \mathsf{else}\ C$ then
        $\beta_1 = \mathsf{instant}(C(s), B), \beta_2 = \mathsf{instant}(C(s), C), S' = S \cup \{s_1, s_2\},$
        $R' = R \cup \{R(s, s_1), R(s, s_2)\} C(s_1) = C(s) \sqcup \beta_1, C(s_2) = C(s) \sqcup \beta_2,$
        $T(s_1) = (T(s) \backslash \{l_{\mathrm{cond}}\}) \cup \mathsf{follows}(()B), T(s_2) = T(s) \backslash \{l_{\mathrm{cond}}\}) \cup \mathsf{follows}(()C)$
    if $A \equiv B \parallel C$ then
        $S' = S \cup \{s'\}, R' = R \cup \{R(s, s')\},$
        $C(s') = C(s) \sqcup \mathsf{instant}(C(s), B) \sqcup \mathsf{instant}(C(s), C),$
        $T(s') = (T(s) \backslash \{l_{\mathrm{par}}\}) \cup \mathsf{follows}(B) \cup \mathsf{follows}(C)$
    if $A \equiv \exists x B$ then
        $S' = S \cup \{s'\}, R' = R \cup \{R(s, s')\}, C(s') = C(s)[y/x] \sqcup \mathsf{instant}(C(s), B),$
        $T(s') = (T(s) \backslash \{l_{\mathrm{exists}}\}) \cup \mathsf{follows}(B)$
    if $A \equiv p(x)$ then
        $S' = S \cup \{s'\}, R' = R \cup \{R(s, s')\}, C(s') = C(s)[y/x],$
        $T(s') = (T(s) \backslash \{l_p\}) \cup \mathrm{ref}_p$
```

As we have seen, each time an agent is analyzed some action are executed. In the following description we show the intuitions behind the formal definitions:

Stop $S \equiv \mathsf{abort}$. When we find a abort agent, we add no information to the store, insert a self-loop over the new node and remove all the labels since the construction must be concluded.

Tell $S \equiv \mathsf{tell}(c)$. The new information c is introduced into the store and the label associated to S is removed from the labels to be executed.

Choice $S \equiv \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$. This agent leads to a set of corresponding branches in the graph. We introduce at most $m + 1$ branches with $m \leq n$, one for each possible successful ask guard. Note that if a c_i condition is not consistent with the store $C(s)$ then the corresponding branch will not be generated. For each new node s'_i, we define the transition $R(s, s'_i)$ and we define an extra arc $R(s, s_{m+1})$ that corresponds to the case when the store does not entail any condition c_i but the execution of concurrent agents proceed (if there are no concurrent agents or there exists but they cannot proceed, then $s_{m+1} = s$ thus a loop is introduced). Moreover, we do not introduce any additional information into the store and the labels are updated.

Conditional $\mathsf{S} \equiv \mathsf{now}\ \mathsf{c}\ \mathsf{then}\ A\ \mathsf{else}\ \mathsf{B}$. The construction process in this case follows the same idea as for the choice operator: we define two new nodes (s'_1 and s'_2) that correspond to the two possible behaviors. The construction is similar for the two branches. The first branch corresponds to the case where the store entails c. It is introduced into the store the information that the agent A can generate in a single time instant. Also the set of labels is updated. The second branch is defined in a similar way.

Parallel $S \equiv A \parallel B$. When a parallel agent is analyzed, the new node generated depends on the execution of the agents A and B in the present time instant. This means that the new store is defined as the union of the information obtained from the execution of A and B (if it is possible to execute them). Also the set of labels depends on these two agents.

Hiding $S \equiv \exists x\ A$. The behavior of the hiding agent is modelled in the graph construction by the introduction of the necessary renaming of variables in the store $C(s')$.

Procedure Call $S \equiv \mathsf{p}(X_1, \ldots, X_n)$. When a procedure call is reached we finish the process by introducing in s' a reference to the initial node of the tccp Structure for p. If there are more agents that must be analyzed, then we continue by considering the tccp Structure already generated for such clause (with the necessary renaming of variables). We link the current node s with a simplified copy of this piece of structure. The simplification consists in eliminating the branches whose condition is inconsistent with the constraints derived by the other (parallel) agents. Thus, the new node s' depends on the execution of the other concurrent agents and the body of the clause for p.

If there are two (or more) procedure calls in parallel the process is similar and as many nodes as different possible behaviors are generated.

In order to illustrate the construction process, in Figure 8.7 we present the construction of the tccp Structure for the program in Figure 8.1. Remember that this program simply detects if the door is open when the microwaves works and in that case turn-off the system and emits an error signal.

We can see how, for the first time instant, two nodes corresponding to the two possible behaviors of the conditional agent have been generated in the specification (n1 and n2). Now look at the node n1 where we have that $L(\mathsf{n1}) = \{\mathsf{lt1}, \mathsf{lt2}, \mathsf{lp}\}$. This means that in order to continue with the graph construction we have to try to execute the agents associated to such labels. The two tell agents update the store with the information that an error combination has been encountered and in the next time instant a stop signal will be present. This is important because when we try to execute the procedure call associated to lp, only one of the two possible branches can be followed.

When we generate new nodes and the corresponding connecting arcs we should consider formulas which are renamed apart. For the sake of simplicity we do not show explicitly the renamings applied in the construction of our example graph. Note that if we find a node equal to another one, a cycle will be formed in the graph and the construction following this branch will terminate.

8.4 Correctness and Completeness

In this section we prove the correctness and completeness of the automatic construction of the model. We first introduce a formal definition for the function which takes the information from the states of the tccp Structure. We define st the set of sequences of the form $\{t \mid t = c_1 \cdot c_2 \cdots c_n \cdots$ where c_i is a conjunction of AP.

Definition 8.4.1 *Given a tccp Structure T, we define the function $\delta : \mathsf{tr}(T) \rightarrow \mathsf{st}$ as follows:*

$$\delta(s) = \begin{cases} C(s_0) & \text{if } s = s_0, \\ C(s_0) \cdot \delta(s') & \text{if } s = s_0 \cdot s', \end{cases} \tag{8.4.1}$$

where $\mathsf{tr}(T)$ is defined following (8.2.1). *The extension of δ over set of sequences is made in the obvious way.*

Now with the following theorem, we show that the above graph construction is correct and complete.

Theorem 8.4.2 *Let T be the tccp Structure corresponding to the tccp specification S. Then the construction T is correct since*

$$\delta(\mathsf{tr}(T)) \equiv [\![S]\!]$$

where δ is the function introduced in Definition 8.4.1, $\mathsf{tr}(T)$ is defined following (8.2.1) *and $[\![S]\!]$ is the operational semantic of the tccp specification S.*

Proof. First of all we define the equivalence relation $\sim$ between configurations of the operational semantics presented in Figure 5.2 and graph states. We identify $\sigma(\Gamma)$ with the store in the configuration Γ. In the graph this store is represented by the function C. Then, we say that a configuration Γ corresponds to a tccp state s if $C(s) \vdash \sigma(\Gamma)$ and $\sigma(\Gamma) \vdash C(s)$. A path in the tccp Structure will correspond to a trace in the operational semantics if each state of the path has a corresponding configuration in the operational semantics and such corresponding configurations form a trace in the semantics.

We have to prove that all paths in the tccp Structure generated from the specification T have an equivalent trace in the operational semantics of T and viceversa. We proceed by structural induction on the language constructs and the length of the path.

Tell $S = \mathsf{tell}(c)$. We have $\pi \in tr(T)$ with $\pi = s_0, s_1, \ldots$, $C(s_0) = d$ and $T(s_0) = \{l_{\mathrm{tell}}\}$. On the other side, we have the trace $\gamma \in [\![S]\!]$ with $\gamma = \gamma_0, \gamma_1, \ldots$ and $\gamma_0 = \langle S, d\rangle$. Assume that s_0 and γ_0 corresponds. By the definition of the construction of the structure and the operational semantics we have that $C(s_1) = \{c \cup d\}$, $T(s_1) = \{\}$ and $\gamma_1 = \langle \emptyset, c \cup d\rangle$ which correspond, thus $\delta(s) \sim [\![S]\!]$.

Choice $S = \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$. We have $\pi \in tr(T)$ with $\pi = s_0, s_1, \ldots$, $C(s_0) = d$ and $T(s_0) = \{l_{\mathrm{choice}}\}$. Then, we have the trace $\gamma \in [\![S]\!]$ with $\gamma = \gamma_0, \gamma_1, \ldots$ and $\gamma_0 = \langle S, d\rangle$. Assume that $s_0 \sim \gamma_0$. By definition of the construction of the structure and the operational semantics we have two cases: the first case is when there is no c_i such that $d \vdash c_i$, then in the construction of the tccp Structure there will be a loop, thus $s_1 = s_0$ whereas in the operational semantics there is no possible transition. The second case is when there exists a c_i such that $d \vdash c_i$. This means that we have $C(s_1) = \{d\}$ and $T(s_1) = \{l_{A_i}\}$. On the other side, we have $\gamma_1 = \langle A_i, d\rangle$. By the (structural) inductive hypothesis, $\delta(A_i) = [\![A_i]\!]$ thus, since s_1 and γ_1 correspond in both the cases, we derive that $\delta(S) = [\![S]\!]$.

Conditional $S = \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$. We have that $\pi \in tr(T)$ with $\pi = s_0, s_1, \ldots$, $C(s_0) = d$ and $T(s_0) = \{l_{\mathrm{cond}}\}$. On the other side, we have the trace $\gamma \in [\![S]\!]$ with $\gamma = \gamma_0, \gamma_1, \ldots$ and $\gamma_0 = \langle S, d\rangle$. Assume that $s_0 \sim \gamma_0$. By definition of the construction of the structure and the operational semantics we have two possible behaviors. In the first case $C(s_1) = \{d \cup \rho(A)\}$[4] and $T(s_1) = \mathsf{follows}(A)$. On the other side, we have $\gamma_1 = \langle A', d'\rangle$ where A' is the agent reached by the execution of A and d' the new store with the information added by the execution of A. By the (structural) inductive hypothesis, $\delta(A) = [\![A]\!]$ thus, since s_1 and γ_1 correspond, we derive that $\delta(S) = [\![S]\!]$. The case when $d \not\vdash c$ is similar to this one.

Parallel $S = A \parallel B$. We have $\pi \in tr(T)$ with $\pi = s_0, s_1, \ldots$, $C(s_0) = d$ and $T(s_0) = \{l_{\mathrm{par}}\}$. On the other side, we have the trace $\gamma \in [\![S]\!]$ with $\gamma = \gamma_0, \gamma_1, \ldots$ and $\gamma_0 = \langle S, d\rangle$. Assume that $s_0 \sim \gamma_0$. By definition of the construction of the structure and the operational semantics we have that $C(s_1) = \{d \cup \rho(A) \cup \rho(B)\}$ and $T(s_1) = \{\mathsf{follows}(A) \cup \mathsf{follows}(B)\}$. On the other side, we have $\gamma_1 = \langle A' \parallel B', d'\rangle$ where A' (B') is the agent reached by the execution of A (B) and d' is the new store with the information added by the execution of A and B. By the (structural) inductive hypothesis, $\delta(A) = [\![A]\!]$ and $\delta(B) = [\![B]\!]$. Thus, since s_1 and γ_1 correspond, we derive that $\delta(S) = [\![S]\!]$.

Exists $S = \exists x\, A$. We have $\pi \in tr(T)$ with $\pi = s_0, s_1, \ldots$, $C(s_0) = d$ and $T(s_0) = \{l_{\mathrm{exists}}\}$. On the other side, we have the trace $\gamma \in [\![S]\!]$ with $\gamma = \gamma_0, \gamma_1, \ldots$ and

[4] When we write $\rho(A)$ we abuse of the notation and we mean the store that is produced in one time instant by the execution of the agent A, following the definition.

$\gamma_0 = \langle S, c\rangle$. Assume that $s_0 \sim \gamma_0$. We know that $C(s_1) = \{d \cup \rho(A[y/x])\}$. Note that $\rho(A[y/x])$ represents the information generated in one time step by the agent $A[y/x]$ which is the result to the application of the substitution y/x to the agent A and $T(s_1) = \mathsf{follows}(A)$. y is a fresh variable, thus the information generated by A involving such variable will not affect the rest of the system.

Now, following the operational semantics we know that $\gamma_1 = \langle \exists^{e'} xB, c \sqcup \exists_x e'\rangle$ and we also know that $\langle A, \exists_x c\rangle \to \langle B, e'\rangle$. Thus, we can identify e' with the information generated from the agent A. $\exists_x e'$ corresponds to the information generated A which is visible for the concurrent agents.

By the (structural) inductive hypothesis, $\delta(A) \subseteq [\![A]\!]$ and $\delta(B) \subseteq [\![B]\!]$. Thus, since s_1 and γ_1 correspond, we derive that $\delta(S) \subseteq [\![S]\!]$.

Procedure Call $S = p(X)$. Let $\pi' = s_1, s_2, \ldots$ be the postfix of $\pi \in tr(T)$ with $\pi = s_0, s_1, \ldots$ and $C(s_0) = d$ and $T(s_0) = \{l_p\}$. Let $\gamma' = \gamma_1, \gamma_2, \ldots$ be the postfix of $\gamma \in [\![S]\!]$ with $\gamma = \gamma_0, \gamma_1, \ldots$ and $\gamma_0 = \langle S, d\rangle$. We have that $s_1 = n$ where n is the first node of the tccp Structure constructed for $p(X)$. We have that $C(s_1) = C(s_0)$ and $T(s_1) = l_A$ where $p(X) :: A$ is a clause in the program. On the other side we have that $\gamma_1 = \langle A, d\rangle$. We have that the length of π' and γ' is $n - 1$, so by the inductive hypothesis we have that $\delta(A) = [\![A]\!]$. Since we have shown that the node and the configuration obtained in one step correspond and are the initial node and state of π' and γ', then we have that $\delta(S) = [\![S]\!]$.

Thus, we have shown that if $\gamma \equiv \langle A, c\rangle \longrightarrow \langle A', c'\rangle \equiv \gamma'$ and the state s and the configuration γ correspond, then $\exists\ R(s, s')$ and s' and γ' correspond. ∎

8.5 The tcc approach

As we have said before, in [FPV00a, FPV00b] was introduced a method to apply the model checking technique to the tcc language. Actually, in those works was defined the modelling phase in detail, giving only a brief description of the specification and the algorithmic phase. In Section 5.4 we have described the main differences between these two languages.

In this section we confront the structure defined to model the behavior of tcc programs (*tcc Structure*) and the tccp Structure. The tcc Structure was defined as follows:

Definition 8.5.1 *Let* AP *be a set of atomic propositions. A* tcc *Structure M over* AP *is a 5-tuple* $M = (S, S_0, T, R, L)$, *where*

1. *S is a finite set of states.*

2. $\mathsf{S}_0 \subseteq \mathsf{S}$ *is the set of initial states.*

3. $\mathsf{T} = \{\mathsf{t}, \mathsf{n}\}$ *is the set of possible* type *of transitions.* t *denotes a temporal transition while* n *denotes a normal transition.*

4. $\mathsf{R} \subseteq \mathsf{S} \times \mathsf{S} \times \mathsf{T}$ *is a transition relation that must be total, that is, for every state* $\mathsf{s} \in \mathsf{S}$ *there is a state* $\mathsf{s}' \in \mathsf{S}$ *such that* $\mathsf{R}(\mathsf{s}, \mathsf{s}', \mathsf{t})$ *or* $\mathsf{R}(\mathsf{s}, \mathsf{s}', \mathsf{n})$.

5. $\mathsf{L} : \mathsf{S} \rightarrow 2^{\mathsf{AP}}$ *is a function that labels each state with the set of atomic propositions true in that state.*

The reader can observe how in this structure there are two kind of transitions: the *timed transitions* and the *normal transitions.* The set of states of the tcc Structure were defined in a similar way as for the tccp Structure and can also be seen as sets of classical states for a Kripke Structure.

However, classical model checking algorithms cannot be applied to tcc Structures. First of all because tcc Structures have two kind of transitions, and secondly because the algorithms cannot handle the notion of state of the graph structure. Note that in the tccp approach we have only one kind of transition relation, thus we have only one problem: how to handle states.

Another main difference between the tcc and the tccp Structure lies in the interpretation of branching points. Branching points in tcc Structures are caused by the interleaving nature of the model. The *normal* transitions are instantaneous in the sense that do not cause the pass of time. The branching points of the tccp Structure due to conditional agents can be viewed as the branching points which could appear in the quiescence points of the tcc Structure, i.e., when passing from one time instant to the following one. However, branching points of the tccp Structure due to Choice agents cannot be identified with anything in the tcc Structure since the tcc model is deterministic.

In [FPV00a, FPV00b] it was defined a method to solve the problems appeared to apply the model checking technique to tcc Structures. In such approach the user must also introduce the time interval over which the verification must be done. Again, such time interval becomes reasonable because the notion of time is defined into the semantics of the language. Using such time interval and the initial value of variables, it was defined a transformation of the tcc Structure into a Kripke Structure. This transformation eliminates the two kind of transitions and unfolds the possible values of variables in the states of the tcc Structure.

The reader can imagine that at this point, the problem was the huge number of states of the transformed structure. Essentially, we loss the possibility to take advantage of the compact representation that can provide the notion of constraint.

As we will see in the next chapters, in the tccp approach it is not necessary to eliminate the kind of transitions (since there is only one type). More important is the fact that it is not necessary to unfold the possible values of variables in order to define a model checking method. Actually, we use a temporal logic which is able to handle the tccp states.

```
construct_ag(input/output $S[]$ : state; input $S_0[], n[]$ : state
             input/output $R$ : relation, $C, T$ : function)
    state :
        st : store;
        $\ell$ : set_of_labels;
    stat1, stat2 : state;
    s[], acc[] : state
    inf[] : store;
    lab[] : set_of_labels;
    j,k,m : int;

    acc = $n$;
    j = 0;
    while acc <> $\emptyset$ do
        stat1 = select(acc);
        acc = remove(acc, stat1);
        inf = instant(stat1.st, stat1.$\ell$);
        lab = follows(stat1.$\ell$);
        m = sizeof(inf);
        for k=1 to m
            if inf[k] <> $\aleph$ then
                s[j] = create_node(inf[k], lab[k]);
                if (s[j] = stat2 $\wedge$ stat2 $\in$ S) then
                    R(stat1, stat2);
                else
                    R(stat1, s[j]);
                    j = j + 1;
                    S = S$\cup${s[j]};
                    acc = acc $\cup$ {s[j]};
```

Figure 8.3: Description of the construction algorithm for agents

```
list_of_sets_of_stores follows($ll$ : label)
        $\ell[]$ : set_of_labels;
        $\ell_1[], \ell_2[]$ : set_of_labels;
        n, i, j : int;

        textsfcase $A$ of          // we assume that $A$ is the agent associated with $ll$.
                abort : $\ell[1] = \{\}$;
                skip : $\ell[1] = \{\}$;
                tell(c) : $\ell[1] = \{\}$;
                $\sum_{i=1}^{n}$ ask($c_i$) $\rightarrow A_i$ : for j = 1 to n
                                                  $\ell[j] = l_{A_j}$;
                                             $\ell[j+1] = \{ll\}$;
                now c then $B_1$ else $B_2$ : $\ell_1$ = follows($l_{B_1}$);
                                                  $\ell_2$ = follows($l_{B_2}$);
                                                  $\ell$ = append($\ell_1, \ell_2$);
                $B_1 \parallel B_2$ : $\ell$ = combine(follows($l_{B_1}$), follows($l_{B_2}$));
                $\exists x\, B_1$ : $\ell$ = follows($l_{B_1}$);
                p(x) : $\ell = \{l_p\}$;          // where $l_p$ represents the label
                                             // of the tccp Structure constructed for p
        end case;
        return $\ell$;
```

Figure 8.4: Description of the auxiliary algorithm follows(ll)

```
list_of_stores instant(input st : store, ll : label)
    s[]: store;
    s_1[], s_2[] : store;
    j: int;
    case A of                // we assume that ll is associated to the agent A
        abort : s[1] = true;
        skip : s[1] = true;
        tell(c) : s[1] = c;
        $\sum_{i=1}^{n}$ ask($d_i$) → $A_i$ : for j = 1 to n
                                    s[j] = {$d_i$};
                                 s[j+1] = {st};
        now d then $B_1$ else $B_2$ : $s_1$ = flat(d, instant(c ⊔ d, $B_1$));
                                    $s_2$ = flat(d, instant(not*(c) ⊔ d, $l_{B_2}$));
                                    s = append($s_1$, $s_2$);
        $B_1$ || $B_2$ : s = combine(instant(c, $B_1$), instant(c, $l_{B_2}$))
        ∃x $B_1$ : s[1] = {c[y/x]} ⊔ instant(c, $l_{B_1}$)  //where y is a fresh variable
        p(x) : s[1] = true;          // where p(x) :: {$l_{B_1}$}$B_1$ is a
                                    // clause of the specification
    end case;
    return s;
```

Figure 8.5: Description of the auxiliary algorithm instant(c, ll)

```
list_of_stores flat(input st : store, ll[] : store)
    s[]: store;
    j, n: int;
    n = sizeof(ll)
    for j = 1 to n
        if ll[j] ⊔ st = false then
            s[j] = ℵ;
        else
            s[j] = ll[j] ⊔ st;
    return s;
```

Figure 8.6: Description of the auxiliary algorithm flat(st,ll)

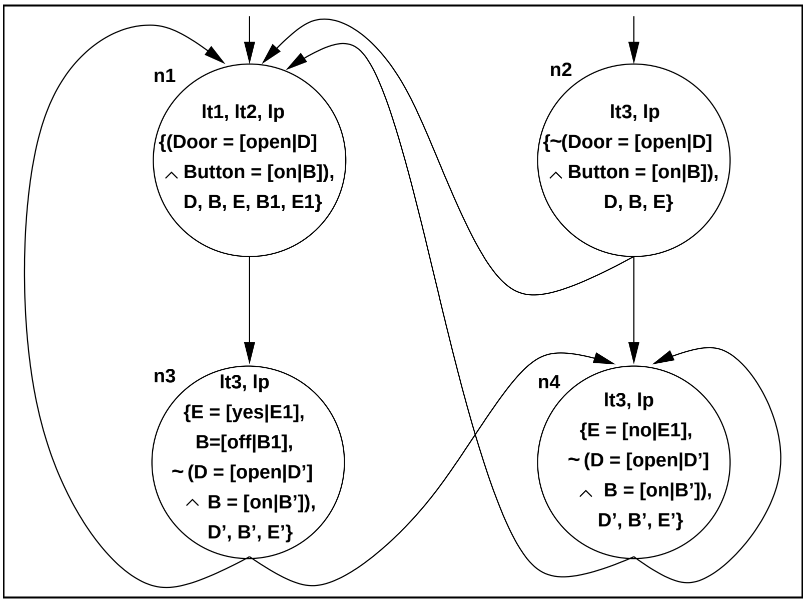

Figure 8.7: Construction of the tccp Structures for the example showed in Figure 5.5

9
The specification of the property

In this chapter we describe in detail the second phase of our model checking approach. Once we have a model that represents the system behavior, we have to specify the property that we want to verify. In many classical approaches (see [CES86, McM93, CGL96]) different temporal logic formalisms are used for such purpose. Some of the most popular logics used for model checking are CTL, CTL* or LTL.

First to introduce the logic used in our approach, we describe briefly the different logics used in the literature and its characteristics in order to compare the logic used in this approach with the classical temporal logics.

9.1 Temporal Logics

Temporal logics were designed by philosophers to study linguistic construct involving the flow of time. Later, in [Pri67], Prior revived the temporal logic theory. The idea was first introduced into the computer science field by Amir Pnueli [Pnu77, Pnu79] to be used to formally verify concurrent systems. In these first works it was showed that it is not sufficient to formally prove properties such as termination in order to guarantee the correctness of a program. Temporal properties such as liveness properties must be also checked (see [Pnu79]).

Temporal logic can be seen as an instance of *modal logic*. Modal logic (see [HC68, Che80]) can be defined as the study of context-dependent properties such as necessity and possibility. The meaning of expressions depends on a specific context. Modal operators are defined in order to combine the different expressions. Thus, the underlying language of modal logic consists of a first-order language extended by some formation rules for modal operators. Usually, the collection of contexts is called the set of *possible worlds*.

For temporal logic, the set of possible worlds can be considered as a linearly ordered set. Usually, semantics of temporal logics are given using Kripke Structures (described in Chapter 1). There is an ordering relation over the set of possible worlds called the *accessibility* relation associated with each modal (or temporal) operator. This relation associates a set of pairs of possible worlds with each modal operator.

There are two main families of temporal logics: *linear time temporal logics* and *branching time temporal logics*. The branching time approach adopts a tree structured time where every time instant may have several immediate successors which correspond to different futures. Linear time logic has a notion of time which is linear, thus each time instant can have only one immediate successor.

The idea of branching time logics was presented in [Abr79]. Then the Unified Branching Time System and the Computational Tree Logic (CTL) logics were defined in [BAMP81, BAMP83] and [CE81, CES83, CES86] respectively. In 1986, Emerson and Halpen gave the definition of CTL* (see [EH83, EH86]). On another way, LTL is the most popular linear time logic. In this section we will briefly present the LTL, CTL and CTL* logics.

For each different logic, different kind of properties can be expressed. Intuitively, using the linear time logic we can express that *something* is true at the current state, in the next state or that it will be true until *other thing* becomes true eventually. We will explain what are these *something* and *other thing* later. On the other side, using the branching time approach we could say that at every state along every computation *something* is false, or that for every computation there exists a state along the computation at which *something* is true.

Computation Tree Logic CTL*. Given a computation tree, the CTL* logic shows all the possible executions starting from an initial state. A computation tree consists in, given a Kripke Structure and an initial state, to unfold the structure into a infinite tree with the initial state as root. The CTL* formulas are composed of *path quantifiers* and *temporal operators.* A path quantifier is used to describe the branching structures in the computation tree. There are two path quantifiers: the **A** quantifier, which denotes "*for all* computation paths", and the **E** quantifier, which denotes "*for some* computation path".

Temporal operators describe properties of paths of the tree. There are five basic operators. The **X** operator denotes that a property is true in the *next time* instant. The **F** operator tells us that a property will be *eventually* true (i.e., in the *future*). The **G** operator specifies that a property holds *always* (*globally*), i.e., it is true at every state on the path. The **U** operator is the *until* operator: given two properties, the until operator says that there exists in the path a state where the second property is true, and at every preceding state the first property holds. Finally, the **R** operator is the dual of the **U** operator: it specifies that the second property holds at every state of the path since (and including) the state where the first property is true. The first property is not required to hold eventually.

Computation Tree Logic CTL. CTL is a restricted subset of CTL* in which temporal operators must be preceded by a path quantifier. Thus, we can say that there are ten basic CTL operators: **AX** and **EX**, **AF** and **EF**, **AG** and **EG**, **AU** and **EU**, and **AR** and **ER**. However, other temporal operators can be defined from these ones. It is easy to see that this logic is less expressive than the CTL* logic (see [CGP99] for example).

Linear Temporal Logic LTL. LTL can be seen as another restricted subset of CTL*. Formulas of the LTL logic are of the form $\mathbf{A}\mathsf{f}$ where f is a path formula whose atomic propositions are the only allowed *state subformulas*[1].

This logic is also less expressive than CTL* but it is not comparable with CTL since there are properties that can be specified using LTL but not CTL and viceversa. To summarize, we can say that CTL* embeds the LTL and the CTL logics but LTL cannot embed CTL and also CTL cannot embed LTL.

There are other temporal logics defined in the literature, for example the ACTL* logic is the restriction to the universal path quantifier of CTL*. The ACTL logic was defined in a similar way from the CTL logic. Moreover, in the last years, interval logics have been used also to verify systems in other approaches providing very interesting results.

9.2 A logic over constraints

In this section we present the logic which we use in our model checking algorithm. This is a temporal logic which has also the ability to handle constraints of a given constraint system. In [BGM01], F. de Boer *et al.* presented a temporal logic for reasoning about tccp programs. In particular, it is an epistemic logic with two modalities, one representing the *knowledge* and the other one representing the *belief*. These two modalities allow us to reason with the input-output behavior of programs.

First of all we have to introduce some notation. There are two modal operators which denote the knowledge and the belief. In particular, given an atomic proposition c of the underlying constraint system, $\mathcal{K}(\mathsf{c})$ and $\mathcal{B}(\mathsf{c})$ are formulas of the logic which mean that c is *known* or c is *belief* respectively. In other words, $\mathcal{B}(\mathsf{c})$ holds if the process assumes that the environment provides c whereas $\mathcal{K}(\mathsf{c})$ holds if the information c is produced by the process itself.

The syntax of temporal formulas for this logic is shown below (see [BGM01] for details):

Definition 9.2.1 *Given an underlying constraint system with set of constraints $\mathcal{C}$, formulas of the temporal logic are defined by*

$$\phi ::= \mathcal{K}(\mathsf{s}) \mid \mathcal{B}(\mathsf{s}) \mid \neg\phi \mid \phi \wedge \psi \mid \exists \mathsf{x}\phi \mid \circ\,(\phi) \mid \phi\,\mathcal{U}\,\psi$$

As for classical temporal logics, it is possible to define other logic operators such as the *always* or *eventually* operators from the basic ones. For example, if we want to express that a formula ϕ is satisfied at some point in the future, we write that

[1] The reader can find the formal definitions of state formula in [CGL96, CGP99]

$\Diamond\phi = \mathit{true}\,\mathcal{U}\,\phi$. To express that a formula ϕ is always satisfied, we can write that $\Box\ (\phi) = \neg(\mathit{true}\,\mathcal{U}\,\neg\phi)$. Moreover, as usually we denote with $\phi \rightarrow \psi$ the formula $\neg\phi \vee (\phi \wedge \psi)$.

We define a *reaction* as a pair of constraints of the form $\langle c, d\rangle$ where c represents the input provided by the environment and d corresponds to the information produced by the process itself. Moreover, it holds that $d \geq c$ for every reaction, i.e., the output always contains the input.

The truth value of temporal formulas is defined with respect to *reactive sequences*. $\langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle$ denotes a reactive sequence which consists of a sequence of reactions. Each reaction in the sequence represents a computation step performed by an agent at time i. Intuitively each pair can be seen as the input-output behavior at time i.

Therefore, given a reactive sequence s we can define the truth values of formulas. We first define the function $\mathsf{first}(s)$ which returns the first reaction of a sequence, i.e., if $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle$ then $\mathsf{first}(s) = \langle c_1, d_1\rangle$. We also define the function $\mathsf{next}(s)$ on reactive sequences which returns the sequence obtained by removing the first reaction of it, i.e., if $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle$ then $\mathsf{next}(s) = \langle c_2, d_2\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle$.

We say that $\langle c, d\rangle \models \mathcal{B}(e)$ if $c \vdash e$, i.e., the reaction "believes" the constraint e if the first component of the reaction (c) entails e. Moreover, $\langle c, d\rangle \models \mathcal{K}(e)$ if $d \vdash e$, i.e., the reaction $\langle c, d\rangle$ "knows" the constraint e if its second component entails e.

Definition 9.2.2 (by F. de Boer *et al.* [BGM01]) *Let s be a timed reactive sequence and ϕ be a temporal formula. Then we define $s \models \phi$ by:*

$$\begin{array}{lll}
s \models \mathcal{K}(c) & \text{if} & \mathsf{first}(s) \models \mathcal{K}(c) \\
s \models \mathcal{B}(c) & \text{if} & \mathsf{first}(s) \models \mathcal{B}(c) \\
s \models \neg\phi & \text{if} & s \not\models \phi \\
s \models \phi_1 \wedge \phi_2 & \text{if} & s \models \phi_1 \text{ and } s \models \phi_2 \\
s \models \exists x\phi & \text{if} & s' \models \phi \text{ for some } s' \text{ s.t. } \exists_x s = \exists_x s' \\
s \models \circ\ (\phi) & \text{if} & \mathsf{next}(s) \models \phi \\
s \models \phi\mathcal{U}\psi & \text{if} & \text{for some } s' \leq s, s' \models \psi \text{ and for all } s' < s'' \leq s, s'' \models \phi
\end{array}$$

where, for a sequence $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle$, we define the existential quantification $\exists_x s = \langle \exists_x c_1, \exists_x d_1\rangle \cdots \langle \exists_x c_n, \exists_x d_n\rangle$.

We say that a formula ϕ is valid ($\models \phi$) if and only if for every reactive sequence s, $s \models \phi$ holds. The reader can see that the modal operators $\mathcal{K}$ and $\mathcal{B}$ are monotonic w.r.t. the entailment relation of the underlying constraint system.

In this thesis we use the logic presented before to reason with tccp programs. Since in such programs the store evolves monotonically we define here the notion of monotonically increasing reactive sequences. Let s be a reactive sequence of the form $\langle c_1, d_1\rangle \cdots \langle c_{n-1}, d_{n-1}\rangle\langle c_n, d_n\rangle$, we say that s is monotonically increasing if it satisfies that $c_i \leq d_i$ and $d_j \leq c_{j-1}$ for each $i \in \{1, \ldots, n-1\}$ and $j \in \{2, \ldots, n\}$. From now we consider only monotonically increasing reactive sequences.

Therefore, whenever a constraint is believed in a specific time instant, then it will be believed also in all the following time instants. Moreover, if a given constraint is known at the present time instant, then it will be known at every time instant in the future. We can express these ideas of monotonicity by the following formulas:

$$\mathcal{B}(c) \rightarrow \square\,(\mathcal{B}(c)) \tag{9.2.1}$$

$$\mathcal{K}(c) \rightarrow \square\,(\mathcal{K}(c)) \tag{9.2.2}$$

In (9.2.1) is expressed the fact that if c is believed at a specific time instant, then it will be always believed. Furthermore, in (9.2.2) we says that if a constraint c is known at a specific time instant, then it will be always known.

Finally, we can define a relation between modal operators. In particular, in (9.2.3) we say that if a constraint c is believed at a specific time instant, then it is also known. Moreover, in (9.2.4) it is expressed that if the constraint c is known at a specific time instant, then it is believed at the following one.

$$\mathcal{B}(c) \rightarrow \mathcal{K}(c) \tag{9.2.3}$$

$$\mathcal{K}(c) \rightarrow \circ\,(\mathcal{B}(c)) \tag{9.2.4}$$

The logic presented in this section can be seen as a kind of linear temporal logic. The reader can see that there are no quantifiers over possible paths. It is considered that each instant of time has only one direct successor. If fact, if we compare this logic with the classical LTL logic (see [CGP99] for example) we can see that each temporal operator corresponds to a temporal operator from LTL.

As we have said before, when we want to reason with programs, we assume (without loss of expressivity) that at each time instant the input information corresponds with the output of the system in the previous time instant. Note that the environment can be modelled as a concurrent process.

9.2.1 Some examples

Here we illustrate which kind of properties we are able to specify using this logic. We use as reference the program example in Figure 5.5. Remember that such example models a very simplified program which controls the state of the door of a microwave.

Since we want to observe only postconditions (see Chapter 5), we can omit the modal operators. In this part of the thesis we reason only with the output of programs, i.e., we consider that all atomic propositions of the formulas presented from now are known in the corresponding time instant.

We could for example check if it is true that when an error is detected, then the microwave has been turned-off. Actually, the error would be occurred in the previous time instant since the door was open and the microwave was working, but the program can emit the error signal only in the following time instant, at the same time that the microwave should be turned-off.

The formula in (9.2.5) represents such property.

$$\neg(\mathit{true}\ \mathcal{U}\neg(\text{Error} = [\text{no} \mid \mathsf{E}] \vee (\text{Error} = [\text{yes} \mid \mathsf{E}] \wedge \text{Button} = [\text{off} \mid \mathsf{B}]))) \tag{9.2.5}$$

It could seem that it is a complicate formula but if we think in terms of the always and eventually operators defined before, it becomes a very intuitive formula:

$$\Box\,(\text{Error} = [\text{no} \mid \mathsf{E}] \vee (\text{Error} = [\text{yes} \mid \mathsf{X}] \wedge \text{Button} = [\text{off} \mid \mathsf{Y}]))$$

We can also model the property that the door will be eventually closed:

$$\Diamond(\text{Door} = [\text{close} \mid \mathsf{D}]) \tag{9.2.6}$$

Let us now remark the importance of the chosen logic in this thesis. We know that states of the tccp Structure represent only partial information. Therefore, if we want to check properties directly in the tccp Structure, then we need a logic able to handle partial information, as is the case of the logic presented in this chapter.

If we use any classical logic, we should consider each possible valuation of the variable values for each tccp State. In that case we had the same problem as in [FPV00a, FPV00b], i.e., we would not take advantage of the compact representation of the system that constraints can provide. Finally, the model checking algorithm would not be applicable by means of the state-explosion problem.

11

The model checking algorithm

The third and last phase of the model checking technique is to define the algorithm which checks if a given temporal formula is satisfied by the model. The idea of the algorithm is similar to that for the classical tableau approach to the LTL model checking problem. The first thing is to construct the *closure* of the formula ϕ that we want to verify.

Actually, we construct the closure of the negated formula $\neg\phi$. We obtain such closure in a similar way as in classical approaches. In our case, atomic propositions are the $\mathcal{K}(\mathsf{c})$ and $\mathcal{B}(\mathsf{c})$ formulas. From this point we consider only the fragment of the logic which uses only the first modality (the knowledge information).

In this work we want to handle properties corresponding to the strongest postcondition semantics, therefore, we can to verify formulas which use only the knowledge modality. To reason with the belief modality we would introduce in the model the information corresponding to the input given to agents. We could also to modify the algorithm in order to allow it to see the information that there were in instant of time before.

When we have the closure of the negated formula, a graph structure (called the *model checking graph*) is constructed. Such structure consists of nodes of the form (q, Φ) where q is a state of the tccp Structure and Φ is a set of formulas from the closure of $\neg\phi$. The constructed graph structure allows us to verify if the property is satisfied or not by the system by using well known graph algorithms. In particular, it is searched a path which starts from an initial state and reaches a *strongly connected component* (SCC) which satisfies some properties. If such path exists, then we can say that the property $\neg\phi$ is satisfied, thus ϕ is not satisfied in the model of the system. In this section we describe all this process more in detail.

Thus, the three main phases of the algorithm are:

1. to calculate the closure of the negated formula $\neg\phi$,

2. to construct the graph which combines the negated formula and the model of the system (the tccp Structure constructed in Chapter 8),

3. to search for a SCC in the model checking graph.

Note that the construction of the graph combining the formula and the model could not terminate. During this construction we use the interval of time which the user provides to the system. This interval imposes a time limit. If such time limit is reached, the system aborts the construction of the graph. The idea is that if this occurs, then we have obtained an over-approximation of the model, thus we could still make some verifications over the finite graph calculated.

11.1 The closure of the formula

First of all, given the formula ϕ we have to compute the closure $\mathsf{CL}(\phi)$. The closure of a formula allows us to determine its truth value. Intuitively, it is the set of subformulas that can affect the truth value. This set is used classically to define tableaux algorithms where subformulas are evaluated in order: simplest formulas are evaluated first, then more complex formulas are considered.

Since $\Diamond$ and $\square$ are defined from the $\circ$ and $\mathcal{U}$ temporal operators, it is sufficient to consider the two last operators. The closure of a specific formula ϕ is composed of formulas whose truth value can determine the truth value of ϕ. Thus, we can say that the closure of ϕ ($\mathsf{CL}(\phi)$) is the smallest set of formulas satisfying the following conditions:

- $\phi \in \mathsf{CL}(\phi)$,
- $\neg\phi_1 \in \mathsf{CL}(\phi)$ iff $\phi_1 \in \mathsf{CL}(\phi)$,
- if $\phi_1 \wedge \phi_2 \in \mathsf{CL}(\phi)$, then $\phi_1, \phi_2 \in \mathsf{CL}(\phi)$,
- if $\exists x \phi_1 \in \mathsf{CL}(\phi)$, then $\phi_1 \in \mathsf{CL}(\phi)$,
- if $\circ(\phi_1) \in \mathsf{CL}(\phi)$, then $\phi_1 \in \mathsf{CL}(\phi)$,
- if $\neg\circ(\phi_1) \in \mathsf{CL}(\phi)$, then $\circ(\neg\phi_1) \in \mathsf{CL}(\phi)$,
- if $\phi_1 \mathcal{U} \phi_2 \in \mathsf{CL}(\phi)$, then $\phi_1, \phi_2, \circ(\phi_1 \mathcal{U} \phi_2) \in \mathsf{CL}(\phi)$.

Note that the case for $\neg\circ(\phi_1)$ is necessary to introduce the formula $\circ(\neg\phi_1)$ which cannot be generated by the other rules. Next we show an example to illustrate this fact.

Example 11.1.1

$$\begin{array}{rll}
\mathsf{CL}(\circ(\psi)) = \{ & \circ(\psi) & [\text{by condition } 1] \\
& \neg\circ(\psi), & [\text{from the original formula, by condition } 2] \\
& \psi, & [\text{from the original formula, by condition } 5] \\
& \neg\psi, & [\text{from } \psi\text{, by condition } 2]
\end{array}$$

$\circ\,(\neg\phi)$ [from $\neg\circ\,(\psi)$, by condition 6]

$\}$

The reader can observe that the last formula in the closure cannot be generated (from the original one) by any other rule except the one presented in the sixth point of the definition.

Now we show the example regarding the microwave program example. The formula (9.2.5) for which we calculate the closure is that presented in the previous chapter.

Example 11.1.2

Taking the example showed in Figure 5.5 as reference, we construct closure of the resulting formula from the negation of (9.2.5). Note that we assume $\neg\neg\phi = \phi$. We also change in the obvious way the disjunction operator into a conjunction:

$$\mathit{true}\,\mathcal{U}\,(\neg(\text{Error}=[\text{no}\mid \mathsf{E}])\wedge\neg(\text{Error}=[\text{yes}\mid \mathsf{E}]\wedge\text{Button}=[\text{off}\mid \mathsf{B}])) \tag{11.1.1}$$

Next we show the closure of the formula. Note that the size of the set of formulas in the closure increases polynomially with the size of the formula (meaning the number of operators in the formula).

$$\begin{aligned}
\mathsf{CL}(\chi) = \{&\mathit{true}\,\mathcal{U}\,(\neg(\text{Error} = [\text{no}\mid \mathsf{E}])\wedge\neg(\text{Error} = [\text{yes}\mid \mathsf{E}]\wedge\text{Button} = [\text{off}\mid \mathsf{B}])),\\
&\mathit{true},\\
&\mathit{false},\\
&\neg(\text{Error} = [\text{no}\mid \mathsf{E}])\wedge\neg(\text{Error} = [\text{yes}\mid \mathsf{E}]\wedge\text{Button} = [\text{off}\mid \mathsf{B}]),\\
&\neg(\text{Error} = [\text{no}\mid \mathsf{E}]),\\
&\neg(\text{Error} = [\text{yes}\mid \mathsf{E}]\wedge\text{Button} = [\text{off}\mid \mathsf{B}]),\\
&\neg(\neg(\text{Error} = [\text{no}\mid \mathsf{E}])\wedge\neg(\text{Error} = [\text{yes}\mid \mathsf{E}]\wedge\text{Button} = [\text{off}\mid \mathsf{B}])),\\
&\text{Error} = [\text{no}\mid \mathsf{E}],\\
&\text{Error} = [\text{yes}\mid \mathsf{E}]\wedge\text{Button} = [\text{off}\mid \mathsf{B}],\\
&\text{Error} = [\text{yes}\mid \mathsf{E}],\\
&\text{Button} = [\text{off}\mid \mathsf{B}],\\
&\neg(\text{Error} = [\text{yes}\mid \mathsf{E}]),\\
&\neg(\text{Button} = [\text{off}\mid \mathsf{B}]),\\
&\circ\,(\mathit{true}\,\mathcal{U}\,(\neg(\text{Error} = [\text{no}\mid \mathsf{E}])\wedge\neg(\text{Error} = [\text{yes}\mid \mathsf{E}]\wedge\text{Button} = [\text{off}\mid \mathsf{B}]))),\\
&\neg(\circ\,(\mathit{true}\,\mathcal{U}\,(\neg(\text{Error} = [\text{no}\mid \mathsf{E}])\wedge\neg(\text{Error} = [\text{yes}\mid \mathsf{E}]\wedge\text{Button} = [\text{off}\mid \mathsf{B}])))),\\
&\circ\,(\neg(\mathit{true}\,\mathcal{U}\,(\neg(\text{Error} = [\text{no}\mid \mathsf{E}])\wedge\neg(\text{Error} = [\text{yes}\mid \mathsf{E}]\wedge\text{Button} = [\text{off}\mid \mathsf{B}])))),\\
&\neg(\mathit{true}\,\mathcal{U}\,(\neg(\text{Error} = [\text{no}\mid \mathsf{E}])\wedge\neg(\text{Error} = [\text{yes}\mid \mathsf{E}]\wedge\text{Button} = [\text{off}\mid \mathsf{B}])))
\end{aligned}$$

}

The definition of the closure of the formula completes the first part of the algorithm. As we show in the next section, now we use this set of formulas to construct the model checking graph.

11.2 The model checking graph

Now let us define the model checking graph. Given a formula ϕ of the logic described in Chapter 9, and the tccp Structure T constructed from the specification, the graph $G(\phi, T)$ is defined as follows

Definition 11.2.1 (Model Checking Graph) *Let ϕ be a formula, $CL(\phi)$ the closure of ϕ as defined in Section 11.1 and T the tccp Structure constructed following the algorithm described in Section 8.3. A node n of the model checking graph is formed by a pair of the form $(s_n, \mathcal{Q}_n)$ where s_n is a state of T and $\mathcal{Q}_n$ is a subset of $CL(\phi)$ and the atomic propositions such that satisfies the following conditions:*

- *for each atomic proposition p, $\mathcal{K}(p) \in \mathcal{Q}_n$ iff $p \in C(s_n)$,*
- *for every $\exists_x \phi_1 \in CL(\phi), \exists_x \phi_1 \in \mathcal{Q}_n$ iff $\exists x \phi_1 \in C(s_n)$,*
- *for every $\phi_1 \in CL(\phi), \phi_1 \in \mathcal{Q}_n$ iff $\neg \phi_1 \notin \mathcal{Q}_n$,*
- *for every $\phi_1 \wedge \phi_2 \in CL(\phi), \phi_1 \wedge \phi_2 \in \mathcal{Q}_n$ iff $\phi_1 \in \mathcal{Q}_n$ and $\phi_2 \in \mathcal{Q}_n$,*
- *for every $\neg_\circ (\phi)_1 \in CL(\phi), \neg_\circ (\phi)_1 \in \mathcal{Q}_n$ iff $\circ (\neg)\phi_1 \in \mathcal{Q}_n$,*
- *for every $\phi_1 \mathcal{U} \phi_2 \in CL(\phi), \phi_1 \mathcal{U} \phi_2 \in \mathcal{Q}_n$ iff $\phi_2 \in \mathcal{Q}_n$ or $\phi_1, \circ (\phi_1 \mathcal{U} \phi_2) \in \mathcal{Q}_n$.*

An edge in the graph is defined as follows: there will be an edge from one node $n_1 = (s_1, Q_1)$ to another node $n_2 = (s_2, Q_2)$ iff there is an arc from the node s_1 to the node s_2 in the tccp Structure and for every formula $\circ (\phi)_1 \in CL(\phi)$, $\circ (\phi)_1 \in Q_1$ iff $\phi_1 \in Q_2$.

Intuitively, for each node of the model checking graph, in Φ we have larger consistent set of formulas that is also consistent with the labelling function (the function C) of the tccp Structure. Moreover, two nodes of the graph are related each other if the temporal formulas in their Φ sets are consistent.

Next we show an example to illustrate how are constructed the nodes of the model checking graph. Actually, we construct the graph for the negation of the property since we intend to prove that there is no computation of the system which satisfies the negated property. This is an equivalent problem to prove that the property is satisfied for all the computations.

Example 11.2.2

In this example we show some nodes of the graph which would result from our program example. We take the tccp Structure showed in Figure 8.7 and the closure set of the formula showed in the previous section.

For each node s_i of the tccp Structure many nodes are generated in the model checking graph. All these nodes have as first component the state s_i and the second component consists of the different consistent sets of formulas derived from $C(s_i)$ and the closure of the formula. Note that there could be more than one consisten set of formulas from the closure. Here we show two of the nodes generated for s_1 and one of the nodes generated for s_2.

$n_1 = (s_1, Q_1)$ where

$Q_1 = \{$

$\mathrm{Door} = [\mathrm{open} \mid D] \wedge \mathrm{Button} = [\mathrm{on} \mid B]$,

true, $\mathrm{Error} = [\mathrm{no} \mid E]$,

$\neg(\mathrm{Button} = [\mathrm{off} \mid B])$,

$\neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B])$,

$\neg(\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]))$,

$\circ\,(\mathit{true}\,\mathcal{U}\,(\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B])))$,

$\mathit{true}\,\mathcal{U}\,(\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]))$

$\}$

$n_2 = (s_1, Q_2)$ where

$Q_2 = \{$

$\mathrm{Door} = [\mathrm{open} \mid D] \wedge \mathrm{Button} = [\mathrm{on} \mid B]$,

true, $\mathrm{Error} = [\mathrm{yes} \mid E]$,

$\neg(\mathrm{Button} = [\mathrm{off} \mid B])$,

$\neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B])$,

$\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B])$,

$\mathit{true}\,\mathcal{U}\,(\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]))$,

$\circ\,(\mathit{true}\,\mathcal{U}\,(\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B])))$

$\}$

$n_3 = (s_2, Q_3)$ where

$Q_3 = \{$

$\mathrm{Error} = [\mathrm{yes} \mid E]$, $\mathrm{Button} = [\mathrm{off} \mid B]$,

$\neg(\mathrm{Door} = [\mathrm{open} \mid D] \wedge \mathrm{Button} = [\mathrm{on} \mid B])$,

true,

$\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]$,

$\neg(\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]))$,

$\circ\,(\mathit{true}\,\mathcal{U}\,(\neg(\text{Error} = [\text{no} \mid \mathsf{E}]) \wedge \neg(\text{Error} = [\text{yes} \mid \mathsf{E}] \wedge \text{Button} = [\text{off} \mid \mathsf{B}]))),$

$\mathit{true}\,\mathcal{U}\,(\neg(\text{Error} = [\text{no} \mid \mathsf{E}]) \wedge \neg(\text{Error} = [\text{yes} \mid \mathsf{E}] \wedge \text{Button} = [\text{off} \mid \mathsf{B}]))$

}

Then, following the definition of the model checking graph, we can define an arc from n_2 to n_3 since for each formula of the form $\circ\ (\phi)$ in the closure, if it is in Q_2 then ϕ is in Q_3.

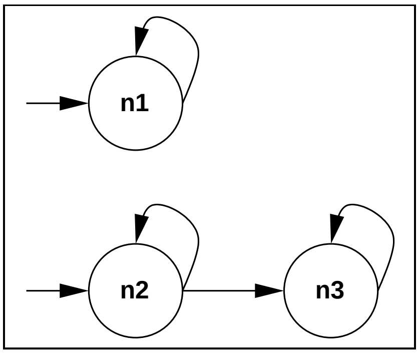

Figure 11.1: A part of the model checking graph for the tccp Structure showed in Figure 8.7 and the formula 9.2.5

In the example, a brief time interval is sufficient to have the complete graph. During the construction we annotate how many steps are needed to reach each node from a root note, which determines the current instant of time. If such instant of time is equal to the time limit, then the construction is aborted and the graph obtained since that moment is given as output of the algorithm.

11.3 The searching algorithm

We must now define the algorithm which can be applied to the constructed model checking graph. This algorithm tells us if the system satisfies or not the formula. We follow the idea of the classical approach: it is sufficient to prove that there is no path satisfying the negation of the property. Thus, in order to verify the formula ϕ, we construct the model checking graph using the negation of ϕ and the model of the system. Then we search for a sequence such that starting from the initial node of the graph, it goes to a *self-fulfilling strongly connected component* (SCC). Next we give the formal definitions of SCC and self-fulfilling SCC.

Definition 11.3.1 (Strongly Connected Component) *Let* G *be a graph, we define a Strongly Connected Component (SCC in short)* C *as a* maximal *subgraph of*

G *such that every node in* C *is reachable from every other node in* C *along a directed path entirely contained within* C.

We say that C *is* nontrivial *iff either it has more than one node or it contains one node with a self-loop.*

Then we can define a kind of strongly connected component. Actually, we will search for SCC satisfying the following properties in our model checking algorithm.

Definition 11.3.2 (Self-fulfilling SCC) *Given a model checking graph* G*, a self-fulfilling strongly connected component* C *is defined as a nontrivial strongly connected component in* G *which satisfies that for every node* n *in* C *and for every* $\phi_1 \mathcal{U} \phi_2 \in Q_n$*, there exists an node* m *in* C *such that* $\phi_2 \in Q_m$*, and viceversa.*

Now, let G be the model checking graph generated following the steps described in Definition 11.2.1. We say that a sequence is an *eventually sequence* if it is an infinite path in G such that if there exists a node n in the path with $\phi_1 \mathcal{U} \phi_2 \in \mathcal{Q}_n$, then there exists another node n' in the same path reachable from n along the path, such that $\phi_2 \in \mathcal{Q}_{n'}$.

Moreover, we can prove the following result, which says that if we find a self-fulfilling strongly connected component in the corresponding model checking graph, then the property represented by the formula is satisfied by the system. Our problem will be to prove that such self-fulfilling SCC does not exists[1].

Theorem 11.3.3 *Let* ϕ *be a formula,* T *a* tccp *Structure and* $G(\phi, T)$ *the corresponding model checking graph. If there exists a path in* G *from an initial node which satisfies a formula* ϕ *to a self-fulfilling strongly connected component, then the model* T *satisfies the formula* ϕ*.*

Proof. In order to prove this theorem we prove instead an equivalent result. We prove that if there exists an eventually sequence starting at an initial node $n = (s, \mathcal{Q}_n)$ such that the formula ϕ is in $\mathcal{Q}_n$, then the model satisfies the formula ϕ. This result is equivalent to the statement of the theorem since classical results ([CGP99, MP95]) show that there exists an eventually sequence starting at a node $n = (s, \mathcal{Q}_n)$ if and only if there is a path in $G(\phi, T)$ from n to a self-fulfilling SCC. We show that we can extend this result directly to our framework.

Assume that we have an eventually sequence $n_1, n_2, \ldots$ where $n_1 = (s_1, \mathcal{Q}_{n_1})$, $n_2 = (s_2, \mathcal{Q}_{n_2})$, etc., starting with $n_1 = n$. This eventually sequence starts at node n_1 with $\phi \in \mathcal{Q}_{n_1}$. By definition, $\pi = s_1, s_2, \ldots$ is a path in the model T starting at $s = s_1$. We want to show that $\pi \models \phi$. We will prove a stronger result: for every formula ψ in the closure of the formula ϕ ($\psi \in CL(\phi)$) and every $i \geq 0, \pi^i \models \psi$ iff $\psi \in \mathcal{Q}_{n_i}$. We follow the classical notations and by π^i with $i \geq 0$ we mean the suffix of the path π starting from the i-th component: $\pi^i = s_i, s_{i+1}, \ldots$. The

[1] Note that the results considers that the construction of the graph has terminated before reaching the time limit provided by the user

proof proceeds by structural induction over the subformulas. There will be six cases corresponding to the six considered operators of the logic.

1. If ψ is an atomic formula, then by Definition 11.2.1 of node n_i, $\psi \in \mathcal{Q}_{n_i}$ iff $\psi \in C(s_i)$.

2. if $\psi = \exists x \chi$ then $\pi^i \models \psi$ iff $\psi \in C(s_i)$.

3. If $\psi = \neg\chi$ then $\pi^i \models \psi$ iff $\pi^i \not\models \chi$. By the inductive hypothesis, this holds iff $\chi \notin \mathcal{Q}_{n_i}$. By Definition 11.2.1, this guarantees that $\psi \in \mathcal{Q}_{n_i}$.

4. If $\psi = \chi_1 \wedge \chi_2$ then $\pi^i \models \psi$ iff $\pi^i \models \chi_1$ and $\pi^i \models \chi_2$. By the inductive hypothesis, this holds iff $\chi_1 \in \mathcal{Q}_{n_i}$ and $\chi_2 \in \mathcal{Q}_{n_i}$. By Definition 11.2.1 this is true iff $\psi \in \mathcal{Q}_{n_i}$.

5. if $\psi = \circ\,(\chi)$ then $\pi^i \models \psi$ iff $\pi^{i+1} \models \chi$. By the inductive hypothesis this holds iff $\chi \in \mathcal{Q}_{n_{i+1}}$. Since $((s_i, \mathcal{Q}_{n_i}), (s_{i+1}, \mathcal{Q}_{n_{i+1}})) \in R$, the above holds iff $\circ\,(\chi) \in \mathcal{Q}_{n_i}$.

6. if $\psi = \chi_1 \mathcal{U} \chi_2$ then by definition of an eventually sequence, there is some $j \geq i$ such that $\chi_2 \in \mathcal{Q}_{n_j}$. Since $\psi \in \mathcal{Q}_{n_i}$, the definition of a node implies that if $\chi_2 \notin \mathcal{Q}_{n_i}$, then $\chi_1 \in \mathcal{Q}_{n_i}$ and $\circ\ (\psi) \in \mathcal{Q}_{n_i}$. In this case, the definition of the transition relation of G implies that $\psi \in \mathcal{Q}_{n_{i+1}}$. It follows that for every $i \leq k < j$, $\chi_1 \in \mathcal{Q}_{n_k}$. By the inductive hypothesis, $\pi^j \models \chi_2$ and for every $i \leq k < j$, $\pi^k \models \chi_1$. Hence $\pi^i \models \psi$.

   Since $\pi^i \models \psi$, then there exists $j \geq i$ such that $\pi^j \models \chi_2$ and for all $i \leq k < j$, $\pi^k \models \chi_1$. We take the minimum j. By the inductive hypothesis, $\chi_2 \in \mathcal{Q}_{n_j}$ and for every $i \leq k < j$, $\chi_1 \in \mathcal{Q}_{n_k}$. Suppose $\psi \notin \mathcal{Q}_{n_i}$. Since $\chi_1 \in \mathcal{Q}_{n_i}$, by Definition 11.2.1 $\circ\ (\psi) \notin \mathcal{Q}_{n_i}$, which implies that $\circ\ (\neg)\psi \in \mathcal{Q}_{n_i}$. Now by definition of the transition relation of G, $\neg\psi \in \mathcal{Q}_{n_{i+1}}$, and hence $\psi \notin \mathcal{Q}_{n_{i+1}}$. Continuing the argument inductively, we would eventually find $\psi \notin \mathcal{Q}_{n_k}$, which is a contradiction since $\chi_2 \in \mathcal{Q}_{n_j}$.

This proves that if we have an eventually sequence, the model satisfies the formula ϕ. Now we have the classical result that can be applied to the graph G. To search for an eventually sequence, we can search for a path from the initial node n to a self-fulfilling SCC. There are algorithms that implement this search with a complexity linear in the size of the graph and exponential in the size of the formula. ■

Example 11.3.4

In our example, once we have constructed the model checking graph completely, we apply a classical graph algorithm to search for a self-fulfilling strongly connected component.

We can see that for our graph there is no a path satisfying this property, thus we can say that the property is satisfied by the program.

11.4 Complexity and related works

In this section we analyze the complexity of the algorithm. We can see that the method is quite inefficient since it is based on the tableau algorithm for LTL. Note that such algorithm is PSPACE-complete.

However, in this complexity only the second and third phases of the model checking method are included since the first phase, in the classical approach, is performed by hand.

Other related works that use the notion of constraint in order to solve the automatic verification problem for infinite state systems can be found in the literature. For example, in [DP01] and [DP99], the authors introduce a methodology to translate concurrent systems into CLP programs and verify safety and liveness properties over such CLP programs. [EM97] introduces a semi-decision algorithm that uses constraint programming in order to verify 1-safe Petri nets. Actually, whereas in [DP01, DP99], constraints are used as an abstract representation of sets of system states, in [EM97] constraint programming is used for solving linear constraints in the implementation of the algorithm.

Regarding the systems that our approach is able to verify, we have seen that there are basically two main cases. The first one is the case when we are able to verify a system without the limitation on the time interval and the second case is when the time limit is reached. The first case corresponds to systems whose infinite nature comes from the fact that they use variables with an infinite domain. These systems are somehow similar to the ones that can be verified in [DP01] for the properties of safety. However, we don't restrict ourselves to these properties and we consider all properties expressible in our logic. Moreover, in the second case we consider a larger class of systems by using the time interval "approximation". If we reach the limit of time imposed by the user (obviously, if the user provides a too short time interval, then some systems of the first class end up in this second category) then we must stop the construction of the graph G at that point. In this case we can verify the system but we must consider that it is an approximation of the original system. We note that there are some limitations in the tccp language since, for example, tccp is not able to model strong preemptions while [DP01] considers a language which can express this behavior.

12
Hybrid cc language

Default tcc is an extension of the tcc language presented in Chapter 3. As in tcc and tccp, this language is useful to model reactive systems. Default tcc enriches the cc model allowing it to specify strong preemptions. A discrete notion of time is sufficient to model reactive systems; however, if we want to model hybrid systems it is necessary to consider a notion of time which is *dense* because these systems evolve continuously over time.

Hybrid systems are the natural continuous extensions of reactive systems. They are defined as those systems which have a continuous component that evolves along the time but which is controlled by a discrete component. An example of such system could be a device that maintains the temperature within an upper and lower limit, i.e., a thermostat. Such device have a continuous component (the temperature) and a discrete control (the temperature limits). Depending on the actual temperature and the target temperature, the device turn-on or turn-off the air conditioning or the heating.

Hybrid concurrent constraint programming (hcc) [GJS96, GJS98] was developed as a paradigm for the modelling, programming and analysis of hybrid systems. hcc is an extension of Default cc. As for the languages previously presented in this thesis, using this methodology the typical advantages of the declarative paradigm are gained. For example, programming in this model is more intuitive and reasoning with the derived languages is easier, than with imperative languages. Another important property is that the specification is executable, that is *what you prove is what you execute.*

Thus, hcc is a concurrent constraint language augmented with a notion of continuous time and defaults. These are the two main differences between hcc and tcc or tccp.

12.1 Syntax

As the rest of cc languages, hcc is defined parametric to an underlying constraint system. In this case, the constraint system used in the cc model is extended to a continuous constraint system in order to be able to describe the continuous evolution of states. Intuitively, the constraint system is enriched with constraints which are able to describe the *initial value* of continuous variables and the *evolution* of such variables (i.e., the first derivative of the variable).

In Figure 12.1 we can see the original syntax of hcc. As in the previous cases, c denotes a constraint of the underlying constraint system and X is a subset of variables in $\mathcal{V}$.

(Agents)	A ::=	c	– Tell
	|	$\mathsf{if\ c\ then}\ A$	– Positive Ask
	|	$\mathsf{if\ c\ else}\ A$	– Negative Ask
	|	$\mathsf{new\ X\ in}\ A$	– Hiding
	|	A, B	– Parallel Composition
	|	$\mathsf{hence}\ A$	– Hence

Figure 12.1: Syntax for hybrid cc programs

In Figure 12.1 you can observe that the choice agent which represents the non-determinism in the cc paradigm is not present. This means that we consider only the deterministic fragment of cc again[1]. Then a timing construct has been added: the hence agent which behaves as "A *continuously* at every time instant after the current one". We explain this construct more in detail later. The Tell operator simply adds c to the actual store. The Positive Ask says that if c holds now, then it behaves as A in the present time instant. The Negative Ask is the negative version of the Positive Ask since, if c does not hold now, then it behaves as A *in the same time instant*. Note that for tcc the Timed Negative Ask construct must wait for the next time instant in order to execute the agent A. In hcc this is not necessary since hcc considers defaults. Finally, the Hiding operator can be seen as an existential quantification over the variable X in the agent A, and A, B is the Parallel Composition, i.e., the agent which imposes that both A and B are executed concurrently.

Looking at the definitions, we can see that the notion of negative information needed to model for example timeouts is represented by the agent $\mathsf{if\ c\ else}\ A$. The reader can observe that this language allows the programmer to model *strong preemptions* since negative information and execution of the defined actions occur in the same time instant [GJS96]. The notion of time is controlled by the $\mathsf{hence}\ A$ agent. In particular, this agent says that the agent A is executed at each time instant after the current one. Combining this operator with the Positive and Negative Ask agents it is possible to define all sorts of behaviors over time. For example, a typical watchdog can be modelled as follows:

$$\mathsf{new}\ A\ \mathsf{in}\ (\mathsf{hence}\ (\mathsf{if}\ \mathsf{X}\ \mathsf{else}\ A, \mathsf{if}\ a\ \mathsf{then\ always}\ \mathsf{X}))$$

defining $\mathsf{always}\ A = (A, \mathsf{hence}\ A)$.

[1] As for the tcc language, see Chapter 3.

In order to make the Hence construct implementable, in [GJSB95, GJS98] some assumptions are made. The basic intuition behind such assumptions is that most physical systems change “slowly”, with points of discontinuous change followed by periods of continuous evolution. With this in mind the *stability condition* is introduced for *continuous constraint systems*. This condition guarantees that for every pair of constraints c and d there is a neighborhood around 0 in which c entails d either everywhere or nowhere. This condition implies that the store of the program cannot change an infinite number of times during a finite time interval.

The Hiding operator can be problematic as well. In fact, in [GJSB95, GJS98] another restriction is introduced in order to avoid situations in which an infinite number of copies of a single variable would be needed. The authors restrict the variables on which existential quantification can be performed to those variables for which one copy suffices for a continuous evolution. The intuition is that only quantifications over variables for which we can permute the agent for existential quantification and the temporal agent without changing the semantics of the system are allowed.

Intuitively, an *hybrid cc computation* is a sequence of alternated point and interval phases. At each point phase, a deterministic Default cc program is executed, thus we can calculate the store of the system in that specific time instant. In such instantaneous computation, all agents which do not contain a preemption operator (negative ask) must be executed. Then, all preemption agents are executed, until a quiescent point is reached, and finally the temporal agents which are in the store are executed.

The execution of the temporal operators at the end of the point phase corresponds to the passage from the point to the interval phase. In this phase, the store is updated executing the current Default cc program and, at the same time, it is computed the maximum time interval in which the guard conditions of ask agents that we have to execute do not change. Once all the temporal agents are analyzed and a time interval has been calculated the execution moves to the next point phase by following each possible path consistent with the value of the discrete variables (which can change at any time instant, since they may be modified by other processes).

12.2 Operational Semantics

As we have pointed before, with the restrictions imposed to make the language implementable, the execution of a hcc program can be seen as a sequence of *point* and *interval phases* alternately. In a point phase, the Default cc rules are applied in order to obtain a quiescence point. In an interval phase, a new store is calculated using the Default cc rules at the same time. The length of the interval phase is determined as the longest interval for which the status of the asks in the program does not change. Once we have obtained the result we move to another interval point phase.

As in tcc, the store of the program is removed when passing from one interval phase to a point phase (actually we have seen that in the tcc model this occurs when a time step is made). This problem is solved during the implementation of the interpreter and we consider it when we construct the graph structure passing the necessary information to the point phase.

The operational semantics is built on the operational semantics for Default cc (see [SJG96]). Note that in this case we use the same definition of configuration as in [SJG94a], i.e., a configuration is a multiset of agents and $\Gamma, \Delta, \ldots$ denote configurations. Moreover, we denote with $\sigma(\Gamma)$ the constraints in the configuration Γ.

Actually, configurations can be *point configurations* or *interval configurations* depending on the phase where they are. A point configuration is a Default cc configuration. The Default cc agents are executed in a real instant of time representing a discrete change. An interval configuration is a triple (c, Γ, Δ) which models the continuous behavior. c represents the initial tokens (similar to an initial condition). Γ are the agents which are *active* in the interval configuration and Δ accumulates the information necessary to continue the execution.

In Figure 12.2 you can see the operational semantics of the language agents. The semantics is given in terms of three kind of transitions: rule $R_{p \to i}$ defines the transitions from point to interval phases; rule $R_{i \to p}$ describes transitions from interval to point phases and the rest of rules describe transitions within the interval phase. For details on the operational semantics, the reader can see [GJS98]. Moreover, $\to$ is the transition relation of Default cc[2], and $\delta(\Gamma')$ denotes the sub-multiset of agents of the form hence A in Γ'.

The transition relation for the interval phase is indexed by d: the constraint used to evaluate defaults as in [SJG96], and by r: the length of the interval.

Intuitively, $\Gamma \downarrow_r^{c,d}$ verifies that the remaining conditionals (Positive or Negative agents) in Γ' were not enabled at any time during the open interval $(0, r)$. Moreover, the output for any time t in the interval phase depends on the variations (first derivative) of variables, the initial value and t.

12.3 Applications

hcc is able to model hybrid systems. In the literature you can find some examples of systems that have been modelled using hcc. For example, in [GSS95] you can find how a photocopier paper path can be modelled as an hybrid systems using hcc. More details about programming techniques regarding hcc can be found in [GJSB95].

In [BC01] is presented an example of how we can use the hybrid concurrent constraint programming paradigm to model biological systems. In particular is showed

[2] See [GJS98] for the complete definition.

$$R_{p \rightarrow i} : \frac{\Gamma \rightarrow \Gamma'}{\Gamma \xrightarrow{\mathsf{hcc}} (\sigma(\Gamma')_{\mathsf{init}}, \delta(\Gamma'), \emptyset)}$$

$$\text{where}\sigma(\Gamma')_{\mathsf{init}} = \{\mathrm{init}(c) \mid \sigma(\Gamma') \vdash_0 \mathrm{init}(c)\}$$

$$R_{\mathsf{then}} : \frac{c, \mathsf{cont}(\sigma(\Gamma)) \vdash^r c'}{(c, (\Gamma, \mathsf{if}\ c'\ \mathsf{then}\ B), \Delta) \xrightarrow{\mathsf{hcc}}_{d,r} (c, (\Gamma, B), \Delta)}$$

$$R_{\mathsf{else}} : \frac{c, \mathsf{cont}(d) \not\vdash^r c'}{(c, (\Gamma, \mathsf{if}\ c'\ \mathsf{else}\ B), \Delta) \xrightarrow{\mathsf{hcc}}_{d,r} (c, (\Gamma, B), \Delta)}$$

$$R_{\mathsf{hence}} : (c, (\Gamma, \mathsf{hence}\ A), \Delta) \xrightarrow{\mathsf{hcc}}_{d,r} (c, (\Gamma, A), (\Delta, A, \mathsf{hence}\ A))$$

$$R_{\|} : (c, (\Gamma, (A, B)), \Delta) \xrightarrow{\mathsf{hcc}}_{d,r} (c, (\Gamma, A, B), \Delta)$$

$$R_{\mathsf{new}} : (c, (\Gamma, \mathsf{new}\ X\ \mathsf{in}\ A), \Delta) \xrightarrow{\mathsf{hcc}}_{d,r} (c, (\Gamma, A[Y/X]), \Delta)$$

$$\text{where } Y \text{ not free in } c, A, \Delta, \Gamma$$

$$R_{i \rightarrow p} : \frac{\exists d \exists r > 0\ \ (c, \Gamma, \Delta) \xrightarrow{\mathsf{hcc}}{}^{*}_{d,r} (c, \Gamma', \Delta')\ \ d = \sigma(\Gamma')\ \ \Gamma' \downarrow^{d,d}_{r}}{(c, \Gamma, \Delta) \xrightarrow{\mathsf{hcc}} \Delta'}$$

Figure 12.2: Operational semantics of the hcc language

how the interaction between two genes is modelled with a simple program. We show such program in Figure 12.3. The same authors have modelled other examples and remark that hcc is a very appropriate model for this kind of applications (see [Cou01]).

```
x = 0; y = 0;
always{if (y < 0.8) x' = −0.02 * x + 0.01;
       if (y >= 0.8)x' = −0.02 * x;
       y' = 0.01 * x; }
sample(x, y);
```

Figure 12.3: Example: Interaction of genes using hcc ([BC01]).

More recently, in [ERJ$^+$03] it has been used the hybrid cc model to simulate some biological processes. In [BC02], it is argued again the appropriateness of the hcc language to model such processes.

The reader can find another very simple example of hcc program in Figure 12.4 which we will use as a reference problem in Chapter 13. This program models a system that has one continuous variable x whose derivative is 1 and when it reaches the value 1, then it is set to 0 again.

```
(    x = 0,
     hence (if prev(x = 1) then x = 0),
     hence (if prev(x = 1) else dot(x) = 1))
```

Figure 12.4: Example: hybrid cc program

The predicate prev asks for the value of constraints in the point where the phase starts (the limit value of the previous phase) whereas the predicate dot asks for the value of derivatives of the variable passed as argument. Both are defined in the constraint system [GJS98].

13

Hybrid cc Model Checking

In this chapter we consider the problem of formal verification for hybrid systems. In particular our starting point is the declarative language hcc presented in the previous chapter. The idea is very similar to the idea for the tccp approach. We define a graph structure, called the hcc *Structure*, which models the system behavior as the Kripke Structure do in the classical approaches. Although the idea of the modelling phase of the method is the same as for the tccp case, in the case of hcc model checking we do not try to apply a model checking algorithm directly to the graph structure constructed.

In this first approach, once we have the model of the system, we define a transformation of the model into a *linear hybrid automata*. Then we can give it as input to the HyTech model checker.

In this dissertation we only consider linear hcc systems since the HyTech tool can handle only such subclass of hybrid systems. However, the construction of the model has been defined in general for any hybrid system specifiable using the hcc language.

13.1 Modelling

We know that one of the first activities in verifying properties of a system is to construct a formal model for the system. This model should capture those properties that must be verified. As we know, both the reactive and hybrid systems cannot be modelled by their input-output behavior. First of all it is necessary to capture the *state* of the system, i.e., a description of the system that contains the values of the variables in a specific time instant. In addition, we have to model how the state of the system changes when an action occurs (*transition* of the system).

In this section we define the graph structure which is able to capture the behavior of hybrid systems. Such kind of state transition system is inspired by the classical notion of *Kripke Structure*. Later We show how we can obtain *automatically* the model (the tcc Structure) of the system starting from the system specification written in hybrid cc.

The set $\mathcal{V}$ represents the universal set of variables. $V \subseteq \mathcal{V}$ is the set of variables that appear in the program clauses specifying the system properties and describe the *state* of the program in each time instant. Variables in $\mathcal{V}$ are typed, where the type of a variable, such as *boolean*, *integer*, etc., indicates the domain D over which the variable ranges.

We can describe *sets* of states and transitions by first-order formulas essentially as usual (see [MP95, CGP99]). The only difference in our case, is the fact that now our first-order formula representing the transition $\mathcal{R}(V, V', T, A)$ has two extra parameters: T expresses whether the transition corresponds to a passage, either from point to interval phases, or from interval to point phases, or is a normal transition. A is a proposition that tells us which agent is being executed in this transition. A is used during the transformation into linear hybrid automaton.

The main idea of the modelling phase is that a state of our graph structure is a set of constraints that define the value of variables and its evolution over time in the case of real-valued variables. Moreover, in each state there is a set of *labels* that represent the point of execution of the system. Labels are introduced in the original program in a similar way as for the tccp model checking approach. We describe the labelling process in the next subsection.

Definition 13.1.1 *Let C be the set of constraints in the* hcc *syntax and L be the set of all possible labels generated to label the original specification of the system. We define the set of states as $S \subseteq 2^{C \cup L}$*

Now we can define formally our graph (structure) capable of representing the system behavior.

Definition 13.1.2 (Hybrid cc Structure) *Let AP be a set of atomic propositions. An* Hybrid cc Structure *(hcc Structure for short) M over AP is a 6-tuple $M = (S, S_0, T, A, R, L)$, where*

1. *S is a finite set of states.*

2. *$S_0 \subseteq S$ is the set of initial states.*

3. *$T = \{n, p, i\}$ is the set of possible* types *of transitions. n denotes a normal transition while p denotes a transition from a point to the interval phase and i denotes a transition from an interval to a point phase.*

4. *A is a set of propositions representing each kind of agent.*

5. *$R \subseteq S \times S \times T \times A$ is a transition relation that must be total, in the sense that for every state $s \in S$ there is a state $s' \in S$ and an agent A such that $R(s, s', n, A)$ or $R(s, s', p, A)$ or $R(s, s', i, A)$.*

6. *$L : S \longrightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.*

A *path* in M from the state s is defined as an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1}, X, Y)$ holds for all $i \geq 0$. A transition where the third component is n (respectively p and i) is called *n-arc* (respectively *p-arc* and *i-arc*).

13.1.1 Labelling

The notion of label allows us to represent the point of execution in a state. We introduce this information by translating the original specification into a labelled version. This transformation consists in introducing a different label associated to each occurrence of a constructor in the specification.

Below we show the details of this transformation. Let P be a statement, the labelled version P_l of it is defined as follows:

- if $P = \mathsf{c}$ then $P_l = \{l_{\text{tell}}\}\,\mathsf{c}$.
- if $P = \mathsf{if\ c\ then}\ A$ then $P_l = \{l_{\text{then}}\}\ \mathsf{if\ c\ then}\ A_l$.
- if $P = \mathsf{if\ c\ else}\ A$ then $P_l = \{l_{\text{else}}\}\ if\,\mathsf{c\ else}\, A_l$.
- if $P = (A, B)$ then $P_l = \{l_{||}\}(A_l, B_l)$.
- if $P = \mathsf{new}\, X\ \mathsf{in}\ A$ then $P_l = \{l_{\text{new}}\}\ \mathsf{new}\ X\, \mathsf{in}\ A_l$.
- if $P = \mathsf{hence}\, A$ then $P_l = \{l_{\text{hence}}\}\ \mathsf{hence}\, A_l$.

In Figure 13.1 we show the labelled version of the program showed in Figure 12.4. We have yet pointed in the previous chapter that both the prev and the dot predicates are defined in the constraint system underlying the hcc language. prev represents some information which holds in the previous time instant whereas dot represents the evolution of a real-valued variable along the time.

$$
\boxed{\begin{array}{ll}
l_{||}(& l_{t_1}\, X = 0, \\
 & l_{\text{hence}_1}\ \mathsf{hence}\ l_{\text{then}}\ (\mathsf{if\ prev}(X = 1)\ \mathsf{then}\ l_{t_2}\, X = 0), \\
 & l_{\text{hence}_2}\ \mathsf{hence}\ l_{\text{else}}\ (\mathsf{if\ prev}(X = 1)\ \mathsf{else}\ l_{t_3}\mathsf{dot}(X) = 1))
\end{array}}
$$

Figure 13.1: Example: labelled hybrid cc program

13.2 Graph construction

In this section we describe how to derive in an automatic way an hybrid cc Structure $M = (S, S_0, T, A, R, L)$ from the hybrid cc program specification of the system. The resulting structure models the system behavior.

We define the set of states as the set of all combinations of constraints that appear in the system specification (see Definition 13.1.1). We generate states while analyzing the program. Intuitively, in each state a collection of constraints that represents the possible values of the variables in that time instant appears.

The constructed graph can simply be seen as a pictorial counterpart of a hybrid cc Structure, with nodes representing the states and arcs representing the transition. To render the fact that we have three kind of transitions (depending upon the value of the third parameter in R) we use three different kind of lines to draw arcs.

The initial node consists of the empty store and the set of labels corresponding to those instructions that can be executed in the first time instant. Then a series of actions are associated to each kind of operator of the language. The execution of such actions produces the graph while analyzing the specification.

During the construction, we need to use some auxiliary function. In particular, we need the function revision which, given a set of labels and a store, checks which labels are active and which others are disabled.

The point of execution of the program is determined by the set of labels in each state of the structure. These labels are associated to the agents that can be executed in the next step. Each label can be *active* or *disabled*. We say that a label is active if the conditions to execute the operator associated to it are satisfied in the store. A label is said to be disabled if the operator represented by this label cannot be executed in that moment because the store does not entail the necessary conditions.

Moreover, we call *normal* labels those labels that do not cause extension over time. All labels representing a temporal or preemption operation are disabled whenever there exists a normal label active in the whole system (perhaps in another state).

The idea of the process is to analyze the specification and, depending on the execution point and on the kind of agent that is being analyzed, to generate new arcs and nodes. In the point phase all computations are considered as instantaneous, thus there is no an increment of time while the store is calculated. However, the time interval calculated during interval phases is considered as the time elapsed along such phase, thus real-valued variables change its value according to its derivatives. When the execution moves from one interval phase to the following point phase, is updated the time and real-valued variables of the interval phase into the point phase.

In order to make the construction finite, we use a time interval provided by the user. Such interval imposes a constraint on the global execution time of our program. Also for this language, the fact that the notion of time is explicitly introduced in the semantics of the language makes reasonable the use of this time interval.

When a new node is generated there are two possibilities: it is equivalent to another node which has been generated in a previous instant, or the new node is not equivalent to any node generated since that instant.

Thus, we can find for the current point phase a quiescence point at the end of the point phase which had already appeared in a previous time instant. Suppose that such previous quiescent point is represented in the graph by the node n, then we must connect the current node to the same nodes to which n is connected and the construction of such branch terminates. Note that in the case that we finish the construction before the time limit is reached we get a quite strong result. In fact, since we did not abort the construction of the graph we can say that our graph *is complete*.

If the current quiescent point had not appeared already in the graph we must continue the construction generating a new node which will be connected by a p-arc to the current one and which will contain the store and the execution point relative to the initial state of the next interval phase. Note that this construction follows the definition of the operational semantics.

In the case that we reach the time limit provided by the user, the construction is terminated and the model obtained is valid only for the actual verification. Therefore, it cannot be reused for different verifications.

Let us now introduce the auxiliary functions revision, follow and localize which are needed during the construction. They should ease the explanation of the construction algorithm.

It is easy, given a specification and an actual agent, to decide which is the following agent in the specification. In Figure 13.2 we define the function which is used for that purpose. Such function calls the algorithm presented in Figure 13.3 which localize a given agent within a specification. Moreover, the function revision is presented in Figure 13.4. The revision algorithm determines which labels are active and which ones are disabled.

Next we describe the actions performed when we analyze the occurrence of each operator. We call *source* node s the node from which a specific agent is executed.

Tell c. A new node s' is generated where the new information a is added, thus $C(s') = C(s) \cup \{c\}$. Moreover, we define a new arc $R(s, s')$. In order to calculate $L(s')$ the follow function is executed. In this case, there is no agent to be executed after the tell agent. Finally, the revision algorithm is applied to $L(s')$ in order to identify the active and disabled labels.

Positive Ask if c then A. When a Positive Ask agent is analyzed, a new node s' is generated where the previous constraints over variables continue to hold and the condition of the positive ask is added ($C(s') = C(s) \cup \{c\}$). An edge that loops in the source node s is added to the graph in order to model the case when the store does not entail the condition of the Positive Ask. Finally, as in the previous case, the set of labels are calculated by executing the two functions follow and revision. Note that the new node s' is generated only in the case that the resulting store is consistent, otherwise such node is not generated.

Negative Ask if c else A. As for the positive ask agent, we have to model two possible situations. In order to model the case when a is entailed by the store

```
struct  agent :
        label : t_label;
        type : {tell, positive, negative, hiding, parallel, hence, null};
        B1 : agent;
        B2 : agent;

follow(S : agent, e : label, labs : set of labels)
S1 : agent;
b : boolean;
localize(S, e, S1, b);
case S1.type of
        tell : labs = {};
        positive : labs = {S.$B_1$.label};
        negative : labs = {S.$B_1$.label};
        hiding : labs = {S.$B_1$.label};
        parallel : labs = {S.$B_1$.label, S.$B_2$.label};
        hence : labs = {S.label, S.$B_1$.label};
end case;
```

Figure 13.2: Algorithm follow

in s ($C(s)$), a new node s_1 is generated where, $C(s_1) = C(s) \cup \{c\}$. The second possible behavior is modelled by the generation of another new node s_2 where $C(s_2) = C(s)$. The sets of labels for both the new nodes are calculated in the same way as in the previous cases, first calculating the labels by using the function follow, and then by applying revision to determine if each label is active or disabled.

Hiding new X in A. In the graph construction, when a hiding agent is analyzed, a new node is generated where variable X is renamed apart in the agent specification A. The information of the renaming of variables is introduced in an auxiliary structure where we put the agents with the quantified variables renamed apart. Then the functions follow and revision are applied as usual.

Parallel (A, B). Since concurrence in hcc is interpreted as interleaving, the parallel composition agent generates different branches of execution. In the graph construction a new node s' is defined. $L(s')$ contains all the labels corresponding to the execution branches generated by the parallel agent. It is important to note that for this new node s' there will be as many direct descendants as active labels contains $L(s')$. $L(s')$ is calculated as usual by applying the functions follow and revision.

Hence hence A. When this agent is analyzed (i.e., the label associated to it is active since a quiescent point has been reached) a new node s' is generated

```
localize($S$ : agent, $e$ : label, $S'$ : agent, $found$ : boolean)
    $S'_1, S'_2$ : agent;
    $found_1, found_2$ : boolean;
    while $S$.label <> $e$ $\wedge$ $S$.type <> 'null' do
    if $S$.type <> 'parallel' then;
      localize($S.B_1, e, S', found$)
    else
      localize($S.B_1, e, S'_1, found_1$);
      if $found_1 = true$ then
        $found = found_1$;
        $S' = S'_1$;
      else
        localize($S.B_2, e, S'_2, found_2$);
        if $found_2 = true$ then
          $found = found_2$;
          $S' = S'_2$;
```

Figure 13.3: Algorithm localize

which is connected to the current one by a *p-arc* and whose store and label function are computed according to the operational semantics of hybrid cc. Finally, follow and revision are applied.

The i-arcs are introduced when we make the passage from an interval to a point phase. When we finish the analysis, we obtain a graph structured as a sequence of point and interval phases.

The following theorem proves the correctness of our graph construction.

Theorem 13.2.1 *Let* T *be the Hybrid* cc *Structure constructed by the above method from the* hcc *specification* S. *Then the construction* T *is correct since*

$$\delta(T) \subseteq [\![S]\!]$$

where δ *is the function that represents the traces in* T *given by the sequences, starting from the root, of program stores in each* hcc *node in a path and* $[\![S]\!]$ *represents the operational semantic of the hybrid* cc *specification* S *in [GJS98].*

Example 13.2.2

In Figure 13.5, the reader can find the graph construction relative to the program in Figure 12.4 (see Section 13.1.1). The reader can observe that there are three kind of arcs. The arcs depicted with continuous lines correspond to normal arcs, dotted arcs correspond to p-arcs whereas i-arcs are illustrated with a discontinuous line. Moreover, active labels are in bold whereas disabled ones are in normal font. The

```
revision(S : set of agents, $Store$ : set of constraints, $S_{act}$ : set of agents,
    $S_{dis}$ : set of agents)
    e : agent;
    T : set of agents;
    while $E \neq \phi$ do
        select(E, e);
        $E = E \backslash e$;
        if (e[2]! = hence) then
            if entail($Store$, e.$label$) then
                $S_{act} = S_{act} \cup e$
            else
                $S_{dis} = S_{dis} \cup e$;
        else
            $T = T \cup e$;
    if empty($S_{act}$) then
        $S_{act} = T$;
        $S_{dis} = \phi$
    else
        $S_{dis} = S_{dis} \cup T$;
```

Figure 13.4: Algorithm revision

reader can see that two different branches appear when there are two active labels at the same moment.

13.3 Transformation

In this section we show how it is possible to transform a hybrid cc Structure into a *hybrid automaton*. In particular, we handle only linear hybrid systems transforming them into linear hybrid automata. In the following the definition of linear hybrid system is given and it is defined a transformation from hybrid cc Structure into linear hybrid automaton.

There exist already several model checkers which implement verification techniques for (linear) hybrid automata. The reader can consult a very popular tool: the HyTech model checked presented in [GNRR93, HHWT95].

An hybrid system is a discrete program within an analog environment that can be modelled by hybrid automata. An hybrid automaton is a generalized finite-state machine. In addition, hybrid systems allow us to represent that the global state of a system changes continuously with the time.

Works on verification of hybrid systems are usually limited to *linear* hybrid automata (see [ACHH93]). Essentially, in such automata, for each variable the rate

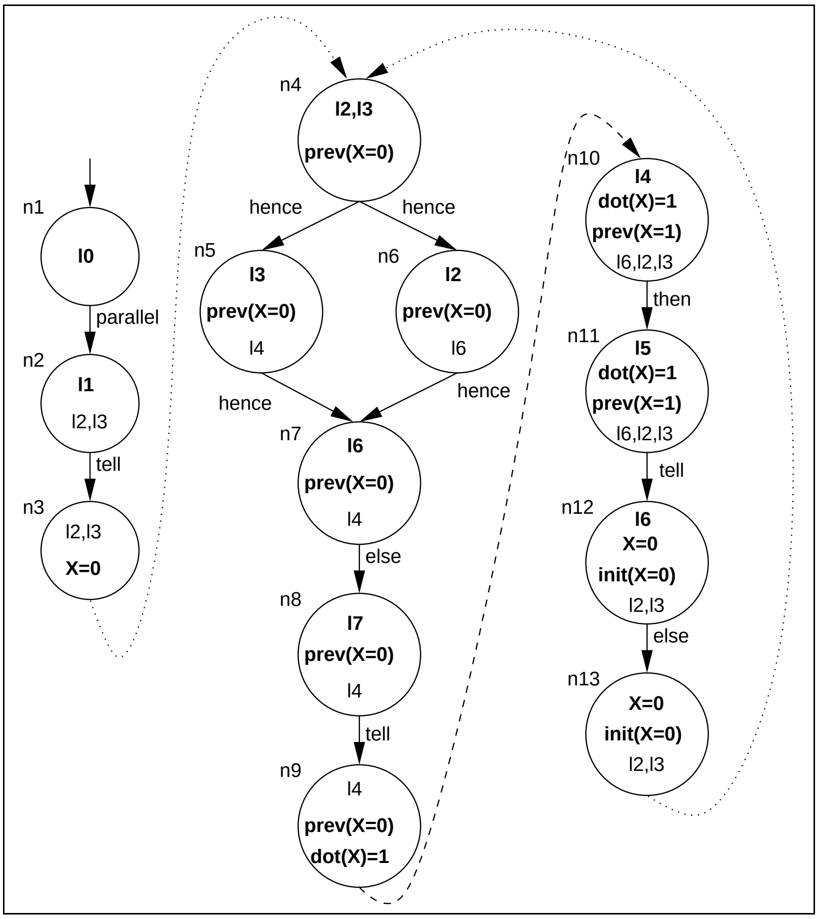

Figure 13.5: Example: Hybrid cc Structure

of change with time is constant and the terms involved in the invariants, guards and assignments are required to be linear.

Next we show the formal definition of a linear hybrid automaton (see [HHWT95] for details). A *linear expression* over a set X of real-valued variables is a linear combination of variables from X with rational coefficients. A *linear inequality* over X is an inequality between linear expressions over X. A *convex predicate* over X is a conjunction of linear inequalities over X; and a *linear predicate* is a disjunction of convex predicates.

Definition 13.3.1 ([HHWT95]) *A* linear hybrid automaton A *consists of the following components.*

Variables. *A finite ordered set* $X = \{x_1, \ldots, x_n\}$ *of real-valued* variables.

Locations. *A finite set* V *of vertices called* locations*, used to model control modes.*

Initial condition. *A state predicate* ϕ^0 *called the* initial condition.

Location invariants. *A labelling function* inv *that assigns to each location* $v \in V$ *a convex predicate* $\mathsf{inv}(v)$ *over* X*, the* invariant *of* v*.*

Transitions. *A finite multiset* E *of edges called* transitions*, used to model discrete events. Each transition* (v, v') *identifies a source location* $v \in V$ *and a target location* $v' \in V$*.*

Instantaneous actions. *A labelling function* jump *that assigns an* update set *and a* jump condition *to each transition* $e \in E$*. The update set* $\mathsf{upd}(e)$ *is a subset of* X*. The jump condition* $\mathsf{jump}(e)$ *is a convex predicate over* $X \cup Y'$*, where* $Y = \{y_1, \ldots, y_k\} = \mathsf{upd}(e)$*, and* $Y' = \{y'_1, \ldots, y'_k\}$*. The primed variable represents the variable value after the transition.*

Urgency flags. *a partial labelling function* asap *that assigns the* urgency flag *ASAP to some transitions in* E*.*

Continuous activities. *A labelling function* rate *that assigns a* rate condition *to each location* $v \in V$*. The rate condition* $\mathsf{rate}(v)$ *is a convex predicate over* $\dot{X}=\{\dot{x}_1, \dot{x}_2, \ldots, \dot{x}_n\}$*. The variable* $\dot{x}_i$ *denotes the rate of change (the first derivative) of* x_i*.*

Synchronization labels. *A finite set* L *of* synchronization labels*, and a labelling function* syn *that assigns a synchronization label from* $L \cup \{\tau_A\}$ *to each transition in* E*. The* internal label τ_A *is specific to the automaton* A*, and does not occur in the label set of any other automaton.*

A *state* (v, s) of the linear hybrid automaton A consists of a location $v \in V$ and a *valuation* of variables $s \in R^n$. The state (v, s) is *admissible* if $s \in \mathsf{inv}(v)$. Control of A may reside in location v only while $\mathsf{inv}(v)$ is satisfied. Only variables in $\mathsf{upd}(e)$ are updated by a transition e.

We have shown that it is possible to construct automatically a model of the behavior of a hybrid system specified by using the hcc language. The model consists in an instance of a hcc Structure, similar to a Kripke Structure. Moreover, we know that it is possible to verify linear hybrid automata by using some popular tools such as the HyTech model checker.

Therefore, we now can outline an algorithm which transforms a (linear) hcc Structure into a linear hybrid automaton. A linear hcc Structure is simply the structure obtained from a linear hybrid system since in this dissertation we consider only such subclass of hybrid systems.

Our first task is to limit the (linear) Hybrid cc Structure using the time interval provided by the user. This is done by assigning values to variables and time intervals, and unfolding possible cycles in the sequence of interval and point phases. With this operation we obtain a representation with a finite number of interval and point phases.

The main idea of this transformation is to identify the interval phases of the hybrid cc Structure with locations in the linear hybrid automaton, and point phases with arcs connecting locations. Obviously, an arc associated with the point phase s_p will join the locations that are associated with the previous and next interval phases, respectively.

Let us now give some details on how we define the components of the linear hybrid automaton. In each location we define the set of variables and derivatives of variables according to the conditions obtained in the relative interval phase. The invariant is the negation of the guard conditions of the ask agents defined in such interval phase. In addition we add a constraint that imposes the maximum time interval that the program can satisfy in such location. This limit is defined by the interval calculated in the interval phase.

Each arc in the linear hybrid automaton corresponds to a point phase in the Hybrid cc Structure. As have said before, an arc joins the two locations that represents the previous and next interval phases of the respective point phase. The set of variables that are updated and their new values are obtained from the store calculated in the point phase. In addition, the label associated to the arc is the label associated to the arc that goes from the point phase to the interval phase. The set of urgent labels is null, because this notion is not used in hcc. Finally, we do not impose restrictions in the synchronization labels and assign a different one to each arc.

In Figure 13.6, the reader can find the linear hybrid automata obtained from the structure in Figure 13.5 associated to the hybrid cc program in Figure 12.4.

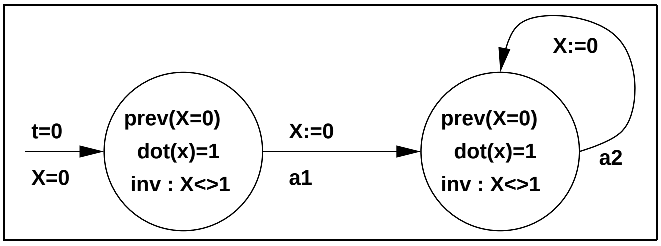

Figure 13.6: Example: linear hybrid automaton

14

Conclusions

In this thesis we have developed and exploited the cc paradigm. We have seen that some languages were defined as extensions of the cc paradigm over time. In particular, in this thesis we have considered the tcc, tccp and hcc languages. The first two languages have a discrete notion of time allowing us to model reactive systems. The last one has a notion of continuous time which allows us to model hybrid systems.

Note that there are other extensions of the cc paradigm. The most recent one is the ntcc language which can be seen as an extension of the tcc model. It introduces the non-determinism into the model.

There are three main contributions in this thesis. First of all we have defined a fully abstract denotational semantics for the tcc model. Secondly we have defined a model checking algorithm for tccp programs, and finally we have defined a first approach to verify (linear) hcc programs.

Denotational Semantics for tcc

We have seen that although both denotational and operational semantics were given when tcc was defined, they do not always coincide. The first contribution of this work is the definition of an appropriate denotational semantics for the tcc language. Our new denotational semantics is fully abstract w.r.t. the operational behavior of the language. The importance of the result presented in this thesis lies in the fact that an appropriate semantics allows us to perform very interesting analysis over languages in a simple way. For example, it is possible to apply some classical static analysis using abstract interpretation.

In particular, we have shown that the problem with the denotational semantics given in the first works lies in the Hiding construct. This construct makes a variable local to a process, but the denotational semantics for this agent is defined as in the cc paradigm where the agent now c else A does not exist, thus the negative information is not taken into account within the denotational semantics.

We can say that the original denotational semantics is an over-approximation of the operational behavior of the language. Actually, the operational and the denotational semantics coincide when we do not combine the Hiding with Negative Ask constructs. This was proved in [NPV02b].

Therefore, we have given a new definition for the denotations of the Hiding agent. Moreover, we have defined the necessary notions in order to handle recursion in the semantics. In particular we have extended the semantics with a notion of environment. We have shown that our new denotational semantics is fully abstract w.r.t. the operational semantics, thus providing a good framework to perform analysis of the tcc language.

We have given the operational semantics of the tcc language in a bit different way from the original definition. However, the two definitions are equivalent. Essentially we have redefined what a configuration is following the idea presented in [PV01] for the ntcc language.

Finally, we have shown a first application of our new semantics to the analysis of the expressivity power of the new construct introduced in tcc to model the timeout or preemption behaviors. This construct is described as now c else A in the language syntax and we have shown that the tcc language becomes a more powerful language than the cc model.

Model Checking for tccp

We think that the second contribution of this thesis is the most significant one. We have exploited the good features of the cc paradigm to solve the classical state explosion problem of model checking. In particular, we have considered the tccp language defined by F. de Boer *et al.* in [BGM00].

We have shown in detail the model checking algorithm for the tccp language. Our goal was to tackle the state explosion problem of model checking and solve it by taking advantage of the features of the language. For example, we use the notion of time defined into the semantics of the language in order to reduce the state space of the model. We also use the notion of constraints underlying the paradigm to construct a compact model of the system and to check properties directly over such model.

Note that the definition of a built-in notion of discrete time makes it reasonable to restrict the model checking task to an *interval* of time, i.e., the interval of time that we want to analyze. Obviously, the user should know and provide the interval on which the verification should be carried out, but this is a reasonable assumption since usually is the specifier who wants to verify the correctness of his software.

In particular, we have defined a graph structure similar to the Kripke Structure, called the tccp Structure. Moreover, we have defined an automatic process to construct a tccp Structure from a tccp program which represents the operational behaviour of the program. Finally we have proved the correctness and completeness of the automatic construction.

Then we have presented the logic defined by F. de Boer *et al.* in [BGM01] which can handle constraints. Essentially, this logic is a linear logic which allows us to check properties directly over the tccp Structure. The algorithm which performs the verification of the system is based on the classic algorithm for the LTL logic.

Moreover we have shown that the verification algorithm is correct.

Note that we have taken advantage of the constraint notion in the different defined phases of the model checking technique. First of all we have used it in order to define what a state is in the tccp Structure. The key idea is that a state of the tccp Structure can be seen as a conjunction of constraints, i.e., as the possible values which the variables of the system can take. We have also seen that a state of the tccp Structure can be viewed as a set of states of a Kripke Structure.

Furthermore, the constraints are directly used in the logic that we have chosen, thus it is not necessary to transform the tccp Structure into a Kripke Structure. Note that classical temporal logics cannot handle tccp Structures directly since such structures have not a complete (or explicit) information of the value of variables.

We have also shown in an intuitive way that our approach is not able to verify *all* infinite-state systems, but only a subclass of such systems. We can verify a very similar subclass of infinite-state systems as [DP99, DP01] can. However, for the other infinite-state systems, we are able to verify an over-approximation of the system thanks to the limitation over time intervals of our method.

To the best of our knowledge this is the first model checking algorithm for systems specified with the tccp language. As future work we intend to study some case studies in order to determine the appropriateness of this method instead of the classical ones. Moreover we have to study formally which class of systems we are able to verify without the time interval restriction.

Model Checking for hcc

The third main result presented in this thesis is the definition of a method to verify hcc programs. We have seen that the hcc language allows us to specify hybrid systems in general and linear hybrid systems in particular. The key idea in this case is also to take advantage of the nature of the cc paradigm. The approach presented in this thesis is the first attempt to apply the model checking technique to the hybrid cc language.

In particular, we have defined a graph structure which represents the behavior of the system specified in hcc and we have transformed it into a linear hybrid automaton which can be given as input to the HyTech model checker. However, note that the hcc Structure defined is able to model hybrid systems in general.

We have described the automatic algorithm which constructs the hcc Structure automatically from the hcc program. Once we have constructed such structure we have defined a method to transform it into a linear hybrid automaton. This transformation is only possible when we handle linear hybrid systems which are a subclass of hybrid systems.

Verification and Constraints

The verification techniques presented in this thesis are novel approaches. Constraints have been used before in verification frameworks but this is the first time that a language of the cc paradigm has been taken as the specification language of the system.

In [FPV00b] we considered the tcc language to verify embedded systems. In such work was defined in detail the modelling phase sketching briefly the rest of the method. Moreover, we presented the approach for hcc in [FPV01]. Finally, we have presented the method to verify reactive systems specified in tccp in [FV03].

In [DP99] and [DP01] linear constraint programming is used in order to model the system transforming the model checking problem into a constraint solving problem. We have not transformed the model checking problem, but we have defined a model checking algorithm which allows us to verify temporal constraint programs.

Bibliography

[AAB$^+$99] P. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of Infinite-State Systems by Combining Abstraction and Reachability Analysis. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 146–159, Berlin, 1999. Springer-Verlag.

[Abr79] K. Abrahamson. Modal Logics for Concurrent Programs. In G. Kahn, editor, *Semantics of Concurrent Computations*, volume 70 of *Lecture Notes in Computer Science*. Springer-Verlag, 1879.

[ACH$^+$95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[AD90] R. Alur and D. L. Dill. Automata for modelling real-time systems. In M.S. Paterson, editor, *Proceedings of the International Colloquium on Automata, Languages, and Programming*, volume 443 of *Lecture Notes in Computer Scienze*, pages 322–335. Springer-Verlag, 1990.

[AHWT97] R. Alur, T. Henzinger, and H. Wong-Toi. Symbolic analysis of hybrid systems. In *Proceedings of the 37th IEEE Conference on Decision and Control*, 1997.

[BAMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The Temporal Logic of Branching Time. In J. White, R. Lipton, and P. C. Goldberg, editors, *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 164–176. ACM Press, 1981.

[BAMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The Temporal Logic of Branching Time. *Acta Informatica*, 20:207–226, 1983.

[BBF$^+$01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schonoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools.* Springer-Verlag, 2001.

[BC01] A. Bockmayr and A. Courtois. Modeling biological systems in hybrid concurrent constraint programming (Poster). In *Proceedings of the 2nd International Conference on System Biology*, 2001.

[BC02] A. Bockmayr and A. Courtois. Using hybrid concurrent constraint programming to model dynamic biological systems. In *Proceedings of the 18th International Conference on Logic Programming*, volume 2401. Springer-Verlag, 2002.

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

[Bei90] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, NY, USA, Second edition, 1990.

[Bei95] B. Beizer. *Brack-Box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, 1995.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In A. Mazurkiewicz and J. Winkowski, editors, *International Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, Berlin, 1997. Springer-Verlag.

[Ber00] G. Berry. The Foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.

[BG96] B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with infinite State Spaces using QDDs. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102, pages 1–12, Berlin, 1996. Springer Verlag.

[BGM00] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161:45–83, 2000.

[BGM01] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In G. Smolka, editor, *Proceedings of 8th International Symposium on Temporal Representation and Reasoning*, pages 227–233. IEEE Computer Society Press, 2001.

[Bir67] G. Birkhoff. *Lattice Theory.* AMS Colloquium Publication, Third edition, 1967.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer-Verlag, 2000.

[BK85] J.A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[BKPR92] Frank de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. On Blocks: locality and asynchronous communication. In *Proceedings of the REX Workshop on Semantics-Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 73–90. Springer-Verlag, 1992.

[BM65] G. Birkhoff and S. MacLane. *A Survey of Modern Algebra*, volume 25. MacMillan, Third edition, 1965.

[Bry] R. E. Bryant. *Graph-based algorithms for boolean function manipulation.* IEEE Transactions on Computers C-35(8).

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered bynary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of Workshop on Logic of programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Berlin, 1981. Springer-Verlag.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In J. R. Wright, L. Landweber, A. Demers, and T. Teitelbaum, editors, *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 117–126. ACM Press, 1983.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications.

ACM Transactions on Programming Languages and Systems, 8:244–263, 1986.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16:1512–1542, 1994.

[CGL96] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking. *Nato ASI Series F*, 152, 1996.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, Cambridge, MA, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.

[Che80] B. F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, 1980.

[CMCHG96] E. M. Clarke, K. M. McMillan, S. Campos, and V. Hartonas-GarmHausen. Symbolic Model Checking. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer-Verlag, July/August 1996.

[Col00] O. Cole. White-box testing should check every line of code. *Dr. Dobb's Journal*, March 2000.

[Cou99] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, volume I of *Lecture Notes in Computer Science*, pages 253–271. Springer-Verlag, September 1999.

[Cou01] A. Courtois. Modélisation de systèmes biologiques en programmmation par contraintes. Raport de DEA, Univ. Henri Poincaré, LORIA, July 2001.

[CVWY92] C. Courcoubetis, M. Y. Vardi, P. Wolper, and N. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2–3):275–288, October 1992.

[Dam96] D. R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, 1996. PhD thesis.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, October 1992.

[DP99] G. Delzanno and A. Podelski. Model Checking in CLP. In R. Cleaveland, editor, *Proceedings 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 223–239, Berlin, 1999. Springer-Verlag.

[DP01] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer-Verlag, 1980.

[EH83] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never". In J. R. Wright, L. Landweber, A. Demers, and T. Teitelbaum, editors, *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, pages 127–140. ACM Press, 1983.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never". *Journal of the ACM*, 33(1):151–178, 1986.

[EM97] J. Esparza and S. Melzer. Model Checking LTL Using Constraint Programming. In P. Azéma and G. Balbo, editors, *Proceedings of the International Conference on Application and Theory of Petri Nets*, volume 1248 of *Lecture Notes in Computer Science*, pages 1–20, Berlin, 1997. Springer-Verlag.

[Eme90] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B of *Formal Models and Semantics*, chapter 16, pages 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[ERJ$^+$03] D. Eveillard, D. Ropers, H. de Jong, Ch. Branlant, and Bockmayr A. Multiscale Modeling of Alternative Splicing Regulation. In C. Priami, editor, *Computational Methods in System Biology*, volume 2602 of *Lecture Notes in Computer Science*, pages 75–87. Springer-Verlag, 2003.

[ES93] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 463–478. Springer-Verlag, June/July 1993.

[Flo67] R. W. Floyd. Assigning Meaning to Programs. In Schwartz, editor, *Proceedings of the Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[FPV00a] M. Falaschi, A. Policriti, and A. Villanueva. Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language. In *Proceedings of the 2000 Joint Conference on Declarative Programming*, La Habana, Cuba, 2000. University of La Habana.

[FPV00b] M. Falaschi, A. Policriti, and A. Villanueva. Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language - I. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Selected papers from 2000 Joint Conference on Declarative Programming*, volume 48 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000.

[FPV01] M. Falaschi, A. Policriti, and A. Villanueva. Time Limited Model Checking. In G. Delzanno, S. Etalle, and M. Gabbrielli, editors, *Proceedings of International Workshop on Specification Analysis and Validation for Emerging Technologies in Computational Logic (SAVE'01)*, 2001.

[FV03] M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *Theoretical Computer Science*, 2003. Submitted for publication.

[GGBM91] T. Gauthier, P. Le Guernic, M. Le Borgne, and C. Le Maire. Programming real time applications with signal. *Proceedings of the IEEE*, 79(Issue 9), 1991.

[GJS96] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Hybrid cc, hybrid automata and program verification. In Alur, Henzinger, and Sontag, editors, *Hybrid Systems III*, volume 4 of *Lecture Notes in Computer Science*, pages 52–75. Springer-Verlag, 1996.

[GJS98] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1–2):3–49, 1998.

[GJSB95] V. Gupta, R. Jagadeesan, V. A. Saraswat, and D. Bobrow. Programming in hybrid constraint languages. In P. Antsaklis, W. Kohn,

A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[GNRR93] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer-Verlag, June 1990.

[GP99] E. L. Gunter and D. Peled. Path Exploration Tool. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 405–419. Springer-Verlag, March 1999.

[Gra97] B. Grahlmann. The PEP tool. In *Tool Presentations of the 18th International Conference on Application and Theory of Petri Nets*, pages 1–10, Toulouse, France, June 1997.

[Gra99] B. Grahlmann. The state of PEP. In *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 522–526. Springer-Verlag, January 1999.

[GSS95] V Gupta, V. A. Saraswat, and P. Struss. Modeling a Photocopier Paper Path. In *Proceedings of the 2nd IJCAI Workshop on Engineering Problems for Qualitative Reasoning*, 1995.

[HC68] G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic*. Methuen and Co LTD, 1968.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(Issue 9):1305–1320, 1991.

[HHWT95] T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: the next generation. In *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 56–65. IEEE Computer Society Press, 1995.

[HK90] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 69(1):45–59, January/February 1990.

[HMP92] T.A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251. Springer-Verlag, 1992.

[HMT71] L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras Part I.* North–Holland, Amsterdam, 1971.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoa85] C. A. R. Hoare. *Communications Sequential Processes.* Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Massachusetts, USA, 1979.

[JJ91] C. Jard and T. Jéron. Bounded memory algorithms for verification on the fly. In *Proceedings of the 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 191–202. Springer-Verlag, 1991.

[KFN93] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software.* Van Nostrand Reinhold, New York, NY, USA, 1993.

[KMM$^+$97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 424–435. Springer-Verlag, 1997.

[Knu80] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Second edition, 1980.

[KRM$^+$93] G. Kutty, Y. S. Ramakrishna, L. E. Moser, L. K. Dillon, and P. M. Melliar-Smith. A Graphical Interval Logic Toolset for Verifying Concurrent Systems. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 138–153. Springer-Verlag, 1993.

[LGS$^+$95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, USA, January 1885.

[LP97] H. R. Lewis and Ch. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, NEw Jersey, USA, second edition, 1997.

[MAB$^+$94] Z. Manna, A. Anuchitanukul, N. Bjorner, E. Chang, M. Colon, A. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, Stanford, California, USA, July 1994.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, NewYork, 1974.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, SU Fisher Research 511/24, 1989.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems. Safety.* Springer-Verlag, Berlin, 1995.

[Mye79] G. J. Myers. *The Art of Software Testing.* Wiley-Interscience, New York, NY, USA, 1979.

[NPV02a] M. Nielsen, C. Palamidessi, and F. Valencia. On the Expressive Power of Concurrent Constraint Programming Languages. In *Proceedings of the 4th International Conference on Principles and Practice of Declarative Programming*, pages 156–167. ACM Press, 2002.

[NPV02b] M. Nielsen, C. Palamidessi, and F. Valencia. Temporal Concurrent Constraint Programming: Denotation, Logic and Applications. *Nordic Journal of Computing*, 1, 2002.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wencel. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. *Lecture Notes in Computer Science*, 2283, 2002.

[ORS92] S. Owre, J. M. Rushby, and M Shankar. PVS: A Prototype Verification System. In *Proceedings of the 11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Inteligence*, pages 748–752. Springer-Verlag, 1992.

[Pap94] Ch. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, Massachusetts, USA, 1994.

[Pat00] R. Patton. *Software Testing.* Sams, 2000.

[Pel93] D. Peled. All from one, one for all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, June/July 1993.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium Foundations of Computer Science*, pages 46–57, 1977.

[Pnu79] A. Pnueli. The Temporal Semantics of Concurrent Programs. In G. Kahn, editor, *Semantics of Concurrent Computations*, volume 70 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1979.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J. W. de Bakker, W.-P de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, Berlin, Germany, 1986.

[Pri67] A. Prior. *Past, Present and Future.* Oxford University Press, 1967.

[PS00] A. Pnueli and E. Shahar. Liveness and Acceleration in Parametrized Verification. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343. Springer-Verlag, 2000.

[PV01] C. Palamidessi and F. Valencia. A Temporal Concurrent Constraint Programming Calculus. In T. Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, page 302 ff. Springer-Verlag, 2001.

[QS82] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium* *on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–350, Berlin, 1982. Springer-Verlag.

[Sar89] V. A. Saraswat. Concurrent Constraint Programming Languages. In *PhD Thesis, Carnegie-Mellon University*, 1989.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming Languages.* The MIT Press, Cambridge, MA, 1993.

[Sip96] M. Sipser. *Introduction to the Theory of Computation.* PWS, Boston, Massachusetts, USA, 1996.

[SJG94a] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *Proceedings of 9th Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, New York, 1994. IEEE.

[SJG94b] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Programming in Timed Concurrent Constraint Languages. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI*, pages 361–410, Berlin, 1994. Springer-Verlag.

[SJG96] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5–6):475–520, 1996.

[SR90] V. A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proceedings of 17th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, New York, 1990. ACM Press.

[SRP91] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of 18th Annual ACM Symposium on Principles of Programming Languages*, pages 333–352, New York, 1991. ACM Press.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proceedings on the 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer-Verlag, June 1990.

[Val01] F. Valencia. Temporal Concurrent Constraint Programming. In T. Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, page 786 ff. Springer-Verlag, 2001.

[Val02] F. Valencia. *Temporal Concurrent Constraint Programming.* PhD thesis, BRICS, University of Aarhus, November 2002.

[Vg96] The VIS group. VIS: A System for Verification and Synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, July/August 96.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, pages 322–331, Cambridge, Massachusetts, USA, June 1986.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing Series. The MIT Press, 1993.

[Wol87] P. Wolper. On the relation of programs and computations to models of temporal logic. In *Proceedings of the Conference on Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 75–123. Springer-Verlag, Altrincham, England, April 1987.