



HAL
open science

Model checking for TCC calculus

Jaime Arias

► **To cite this version:**

| Jaime Arias. Model checking for TCC calculus. Universidad Javeriana. 2012. hal-04257266

HAL Id: hal-04257266

<https://hal.science/hal-04257266>

Submitted on 25 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Acta de Correcciones al Proyecto de Grado
Ingeniería de Sistemas y Computación e Ingeniería Electrónica

Fecha: Diciembre 14 de 2012

Autores: Jaime Eduardo Arias Almeida

Nombre del Proyecto de Grado: Model Checking for TCC Calculus.

Director: Dr. Carlos Olarte Vega – Dr. Eugenio Tamura Morimitsu

Como indica el artículo 2.27 de las Directrices de Trabajo de Grado, he verificado que los estudiantes indicados arriba han implementado todas las correcciones que los Jurados del Proyecto de Grado definieron que se efectuaran, como consta en el Acta de Calificación correspondiente.

Firma de Director(a) del
Proyecto de Grado

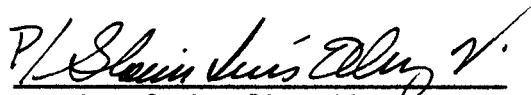
Firma de Director(a) del
Proyecto de Grado

Nota de Aceptación

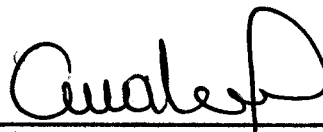
Aprobado por el Comité de Trabajo de Grado en cumplimiento de los requisitos exigidos por la Pontificia Universidad Javeriana para optar el título de Ingeniero de Sistemas y Computación e Ingeniero Electrónico.



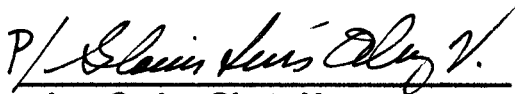
Dr. MAURICIO JARAMILLO AYERBE
Decano Académico de la Facultad de Ingeniería



Ing. Carlos Olarte Vega
Director Carrera de Ingeniería



Ing. Ana Victoria Prados
Directora Carrera de Ingeniería



Ing. Carlos Olarte Vega
Director Trabajo de Grado



Ing. Eugenio Tamura Morimitsu
Director Trabajo de Grado



Ing. Camilo Rueda
Jurado 1



Ing. Jorge Finke
Jurado 2



Moreno Falaschi (Universidad de Siena)
Jurado Externo

MODEL CHECKING FOR TCC CALCULUS

JAIME EDUARDO ARIAS ALMEIDA

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERÍA
INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
INGENIERÍA ELECTRÓNICA
SANTIAGO DE CALI
2012

MODEL CHECKING FOR TCC CALCULUS

JAIME EDUARDO ARIAS ALMEIDA

*Proyecto de grado para optar al título de
Ingeniero de Sistemas y Computación e
Ingeniero Electrónico*

Directores : Dr. Carlos OLARTE
Dr. Eugenio TAMURA

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERÍA
INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
INGENIERÍA ELECTRÓNICA
SANTIAGO DE CALI
2012

Santiago de Cali, Diciembre 10 de 2012

Doctor
MAURICIO JARAMILLO AYERBE
Decano Académico de la Facultad de Ingeniería
Pontificia Universidad Javeriana
Ciudad

Certificamos que el presente trabajo de grado, titulado "MODEL CHECKING FOR TCC CALCULUS" realizado por JAIME EDUARDO ARIAS ALMEIDA, estudiante de Ingeniería de Sistemas y Computación e Ingeniería Electrónica, se encuentra terminado y puede ser presentado para sustentación.

Atentamente,

Dr. CARLOS OLARTE VEGA
Director del Proyecto

Dr. EUGENIO TAMURA MORIMITSU
Director del Proyecto

Santiago de Cali, Diciembre 10 de 2012

Doctor
MAURICIO JARAMILLO AYERBE
Decano Académico de la Facultad de Ingeniería
Pontificia Universidad Javeriana
Ciudad

Por medio de ésta, presento a usted el trabajo de grado titulado "MODEL CHECKING FOR TCC CALCULUS" para optar el título de Ingeniero de Sistemas y Computación e Ingeniero Electrónico.

Espero que este trabajo reúna todos los requisitos académicos y cumpla el propósito para el cual fue creado, y sirva de apoyo para futuros proyectos en la Universidad Javeriana relacionados con la materia.

Atentamente,

JAIME EDUARDO ARIAS ALMEIDA

ARTICULO 23 DE LA RESOLUCIÓN No. 13 DEL 6 DE JULIO DE 1946
DEL REGLAMENTO DE LA PONTIFICIA UNIVERSIDAD JAVERIANA.

“LA UNIVERSIDAD NO SE HACE RESPONSABLE POR LOS CONCEPTOS EMITIDOS
POR SUS ALUMNOS EN SUS TRABAJOS DE TESIS. SÓLO VELARÁ PORQUE NO SE
PUBLIQUE NADA CONTRARIO AL DOGMA Y A LA MORAL CATÓLICA Y PORQUE LAS
TESIS NO CONTENGAN ATAQUES O POLÉMICAS PURAMENTE PERSONALES;
ANTES BIEN, SE VEA EN ELLAS EL ANHELO DE BUSCAR LA VERDAD Y LA JUSTICIA”

Resumen

La Programación Concurrente por Restricciones (ccp) es un formalismo para modelar sistemas concurrentes en el cual agentes (procesos) interactúan con otros agregando (*telling*) y leyendo (*asking*) información representada como restricciones en un medio compartido (*store*). La Programación Concurrente Temporal por Restricciones (tcc), extiende el modelo ccp agregándole constructores temporales para modelar agentes temporales y sistemas reactivos.

La verificación formal cumple un papel muy importante en la detección de errores en sistemas concurrentes, ya que permite determinar si el modelo de un sistema satisface o no una propiedad. Model checking es una técnica de verificación formal que, dado el modelo de un sistema y una propiedad, comprueba sistemáticamente si el modelo satisface o no la fórmula.

Este proyecto de grado investiga la técnica de model checking como un método formal para la verificación de programas tcc. La investigación se lleva a cabo mediante la definición de un algoritmo de model checking para el cálculo tcc. Para lograr esto, nosotros extendemos el algoritmo clásico de model checking para LTL.

Primero definimos una estructura llamada tcc Structure la cual permite modelar el comportamiento de un sistema tcc, además describimos una lógica que permite razonar sobre programas tcc. Luego se presenta el grafo de model checking y las propiedades que debe cumplir para determinar que el modelo satisface la propiedad. Al final, se presenta un prototipo que implementa el algoritmo propuesto.

Palabras Clave: Model checking, programación concurrente temporal por restricciones, verificación formal automática.

Abstract

Concurrent Constraint Programming (**ccp**) is a formalism for concurrency in which agents (*processes*) interact with one another by adding (*telling*) and reading (*asking*) information represented as constraints in a shared medium (*store*). Temporal Concurrent Constraint Programming (**tcc**) extends **ccp** by adding temporal constructs for modeling timed and reactive systems.

Formal verification plays an important role in detecting errors in concurrent systems since it allows to check whether or not a system satisfies a given property. Model checking is a formal verification technique that, given a finite-state model of a system and a property, it systematically checks whether the property is satisfied by the model.

This project degree studies model checking as a formal method for the verification of **tcc** programs. The study is conducted by defining a model checking algorithm for **tcc**. To accomplish this, we extend the classical algorithm of model checking for LTL.

We define a structure called *tcc Structure* which allows to model the behavior of a **tcc** system, and we then describe a specific logic which allows to reason about **tcc** programs. We also introduce the model checking graph and the properties that it must meet to determine that the model satisfies the property. Finally, we present a prototype which implements the proposed algorithm.

Keywords: Automatic formal verification, model checking, temporal concurrent constraint programming.

Acknowledgments

First of all, I want to express my gratitude for my supervisor Carlos Olarte. He has always guided me in the right direction and provided a warm environment to grow as a researcher and as person. Having met Carlos was, to say the least, a very important event in my life. His dedication and enthusiasm for Computer Sciences have inspired me to pursue a research career. I am very fortunate to have worked with him.

I am most grateful to Camilo Rueda for giving the opportunity to belong to the AVISPA group, and fund my research. Also, I want to express my deepest admiration for his outstanding work.

I want also to express my gratitude to the members of the AVISPA group for their important remarks to the work.

I also want to show my affection for my friends Laura Pérez, Andrés Oviedo, Sandra Forero, Jairo Alegría, Daniel Almeida, Anthony Illera, and Leidy Siachoque.

Special thanks to my colleagues at Research Laboratory of the Department of Electronics and Computer Science, Jairo Velasco, Andrés Barco, Jheyson Vargas, Claudia Oviedo, Mauricio Toro, Alejandro Arbeláez, Mario Mora, and Natalia Villegas. Thanks to you I was never bored during my stay in the laboratory.

I thank my friends in the #archlinux-co IRC channel who made my sleepless nights less painful. They are Alejandro Rean, Matias Russitto, Mariano Street, Gustavo Gómez, Julián Camargo, Juan Camilo Noreña, Johan Duarte, Andrés Quintero, and Diego Herrera.

And to everyone who directly or indirectly helped with the development of this work.

I dedicate this work to the two most important persons in my life: my mother María Almeida and my brother Diego Arias. They always encouraged me and showed me their affection.

*Jaime E. Arias Almeida,
December 14, 2012*

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions	3
1.3	Document Structure	3
2	Preliminaries	5
2.1	Process Calculi	5
2.2	Concurrent Constraint Programming	5
2.2.1	Reactive Systems and Timed CCP	6
2.3	Transition Systems	7
2.3.1	Kripke Structure	8
2.4	Linear-Time Temporal Logic	9
2.4.1	Syntax	9
2.4.2	Semantics	11
2.5	Model Checking	11
3	The tcc Language	13
3.1	Constraint System	13
3.2	Syntax	14
3.3	Operational Semantics	15
3.4	Program Example	15
3.5	Summary	16
4	A Model for tcc Programs	17
4.1	Program Labeling	17
4.2	The tcc Structure	18
4.3	Construction of the Model	20
4.4	Simplification of the Model	22
4.5	Construction Example	24
4.6	Summary	24
5	A Logic for the Specification of Properties	29
5.1	The ntcc Logic	29
5.1.1	Temporal Logic Syntax	29
5.1.2	Temporal Logic Semantics	30
5.1.3	Specification Example	31
5.2	Summary	31

6	The Model Checking Algorithm	33
6.1	The Closure of the Formula	33
6.1.1	Closure Example	34
6.2	The Model-Checking Graph	34
6.2.1	Model Checking Graph Example	35
6.3	The Searching Algorithm	41
6.3.1	Searching Algorithm Example	43
6.4	Model Checking Example	44
6.5	Summary	50
7	A Prototypical Tool	53
7.1	Inputs	53
7.1.1	Property	53
7.1.2	System Model	54
7.2	Model Checking Function	55
7.2.1	Closure	56
7.2.2	Model Checking Nodes	56
7.2.3	Model Checking Graph	57
7.2.4	Self-Fulfilling Strongly Connected Components	57
7.3	Summary	59
8	Concluding Remarks	61
8.1	Overview	61
8.2	Future Work	61
	Bibliography	63

Introduction

This degree project studies Model Checking as a formal method for the verification of Temporal Concurrent Constraint (**tcc**) programs. In particular, we propose a structure for modeling the behavior of a system, a temporal logic to reason about the system, and a model checking algorithm to formally verify the system properties. The method proposed is based on the work developed by Falaschi and Villanueva [FV06].

1.1 Motivation

Nowadays concurrent systems are pervasive in several domains and applications. For example, the sciences (e.g. biological systems), the engineering (e.g. communication protocols), and the arts (e.g. tools for computer music).

The previous examples illustrate the practical relevance and complexity of concurrent systems. Therefore, it is crucial to develop computational models which allow to describe, analyze and reason about the behavior of such complex systems. Process calculi are computational formalisms that are commonly used for modeling and reasoning about concurrent systems. The main idea underlying process calculi is the abstraction of systems in terms of basic units known as *processes*. The most representatives are CCS [Mil89], the π calculus [Mil99, SW03] and CSP [Hoa85].

Concurrent Constraint Programming (**ccp**) [SRP91, Sar93] has emerged as a model for concurrency that combines the traditional operational view of process calculi with a declarative one based upon logic. A fundamental feature in **ccp** is the specification of concurrent systems in terms of constraints. A constraint is a first order formula representing partial information about certain variables (e.g. $x + y > 0$). In this model, agents interact with each other by adding (or *telling*) and asking information (*constraints*) in a shared medium (*store*).

Some concurrent systems react continuously with their environment (e.g. biological systems). These systems are known as reactive systems [BG92]. Saraswat, Jagadeesan and Gupta developed the Timed **ccp** [SJG94a, SJG94b] for modeling this type of systems. The fundamental move in the **tcc** model is to extend the standard **ccp** with *delay* and *time-out* operators. Time in **tcc** is conceptually divided into *time intervals* (or *time units*). In a particular time interval, a deterministic **ccp** process receives a stimulus (i.e. a constraint) from the environment, it executes with this stimulus as the *initial store*, and when it reaches its resting point, it responds to the environment with the *final store*. Also, the resting point determines a residual process, which is then executed in the next time interval.

Formal verification plays an important role in detecting errors in concurrent systems since it allows to check whether or not a system satisfies a specific property. Nowadays the early detection

of errors is essential because the presence of a fault in systems could be catastrophic: take for instance air traffic control systems, medical instruments, aircrafts among others. Clarke, Grumberg and Peled [CGP99] show some significant examples how formal verification techniques help to find errors in modeled systems.

There are two important formal verification techniques: *theorem proving* and *model checking*. Theorem proving was the first technique for formal verification. The idea was introduced by Floyd and Hoare in [Flo67, Hoa69]. This is a deductive method which is performed essentially manually, thus it can be very difficult, inefficient and error prone. The success of the verification process depends on the capability of the user. Therefore, this technique must be used by people expert in mathematics and logic. On the other hand, this method is very reliable because it uses mathematics and logic theory, and it can verify large systems since it is not limited by the size of the state space.

The second formal verification technique is *model checking*. This technique was first introduced by Clarke and Emerson [CE82] and Quielle and Sifakis [QS82]. This method checks in a fully automatic way that the execution sequences of the system (i.e. an exhaustive analysis of the state-space) are a model of the formula representing the property. The main problem of this technique is that the *state-space* of a concurrent system can be huge, then the number of states needed to model the system accurately may exceed the amount of available computer memory (*state explosion problem*).

Currently, the AVISPA¹ research group has developed several applications in emergent areas such as security, biology and multimedia interaction using the `tcc` model. However, the verification of these models is performed using inductive techniques since the model does not provide automatic formal verification tools. Therefore, the verification is difficult, error prone and performed only by experts in the field.

This degree project then strives for developing a model checking algorithm for Temporal Concurrent Constraint Programming (`tcc`) calculus. Doing that, we provide the `tcc` model with an automatic formal verification tool which allows to verify systems easily and faster. Moreover, we contribute to the points made by Hubert Garavel [Gar08]:

The times have gone, where formal methods were primarily a pen-and-pencil activity for mathematicians. Today, only languages properly equipped with software tools will have a chance to be adopted by industry. It is therefore essential for the next generation of languages based on process calculi to be supported by compilers, simulators, verification tools, etc. This also applies to new models for concurrency, such as mobile calculi and bigraphs. The research agenda for theoretical concurrency should therefore address the design of efficient algorithms for translating and verifying formal specifications of concurrent systems.

Our approach is based on the work by Falaschi and Villanueva [FV06]. Then, we divide the model checking algorithm into three main phases: modeling, specification and verification.

Firstly, we start by defining a structure that models the behavior of a system. Moreover, we present an algorithm for building the structure from a `tcc` specification. This algorithm does not

¹<http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:avispa>

include the local agent. Although our approach is based on the work done in [FV06], the structure defined here is quite different from the structure presented in [FV06]. Finally, we introduce a method to reduce the number of states of a model in order to mitigate the state explosion problem.

Secondly, we present a temporal logic for reasoning about `tcc` programs. This logic is based on sequences of constraints instead of classical states. Thus, we can check properties directly over the model of the system.

Thirdly, we define an algorithm that determines if the system satisfies the property. This algorithm receives as input the model of the system and the formula to be verified.

Finally, we describe a prototype that implements our model checking algorithm.

This work is part of the REACT PLUS² project of the AVISPA research group. REACT PLUS addresses the development and application of formal methods in real-life systems. It takes the challenging task of developing the underlying theory and machine-assisted tools for verifying concurrent systems

1.2 Contributions

The main contributions associated with this work are presented below:

1. We define a structure called `tcc` Structure which allows to model the behavior of a `tcc` program. Moreover, we specify an algorithm to construct the model from a `tcc` specification.
2. We introduce a method to reduce the size of a `tcc` Structure.
3. We specify a model checking algorithm to verify `tcc` programs.
4. We develop a prototype of the proposed model checking algorithm.

1.3 Document Structure

In what follows we describe the structure of this document. Each chapter concludes with a summary of its content.

Chapter 2 [Background]. In this chapter we introduce the basic concepts and terminology used in this document. We start by describing the Concurrent Constraint Programming model and its temporal extension `tcc`. We then present the notions of transition systems and temporal logic which are important for modeling systems and specify properties. Finally, we explain the model checking technique.

Chapter 3 [tcc Language]. This chapter presents the formal syntax of `tcc` and its operational semantics. Furthermore, this chapter introduces the notion of constraint system which is fundamental to `ccp` based calculi, and it shows a program modeled using this language.

²<http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:react-plus>

Chapter 4 [Model for tcc Programs]. In this chapter we define the structure which allows us to model the behavior of tcc systems. Moreover, we describe how to construct the model from the specification of the tcc system, and we present a procedure to reduce its size in order to mitigate the state explosion problem.

Chapter 5 [Property Specification]. This chapter describes the temporal logic that we use to reason about tcc systems and to express properties of them. This logic has the feature that it is based on sequences of constraints instead of states.

Chapter 6 [Model Checking Algorithm]. In this chapter we present the algorithm which allows us to determine if a formula is satisfied by the model. This algorithm is based on the classical tableau algorithm for the LTL model checking problem. Thus, we define the structure called model checking graph which is essential to the algorithm. At the end of the chapter, we illustrate our algorithm by showing two examples.

Chapter 7 [Prototype]. This chapter introduces a prototype tool that implements the model checking algorithm presented in Chapter 6, and it also describes the auxiliary functions employed. Furthermore, this chapter defines a structure to represent a model of the system and a formula since these are the inputs of the algorithm.

Chapter 8 [Concluding Remarks]. This chapter presents the main results derived from this degree project and gives some directions for future work.

Preliminaries

In this chapter we introduce the basic concepts and terminology used in this document. We briefly describe the Concurrent Constraint Programming model and its temporal extension `tcc`, transition systems, temporal logic and the model checking technique. We do not intent to give an in-depth review of these concepts but rather to contextualize the development of the model checking algorithm in this degree project. We encourage the reader to follow the references to have a complete description of each topic addressed in this chapter.

2.1 Process Calculi

Process calculi [Bae05] are formal methods for reasoning about concurrent systems. The main idea underlying process calculi is the abstraction of real systems in terms of basic units known as processes. The calculi provide precise elements to describe systems as combination of processes, as well as offer tools to study the behavior of systems over time, providing a high level description of interactions, communication, and synchronization between a group of independent agents or processes.

Process calculi in the literature mainly agree in their emphasis upon algebraic semantics. The main representatives are CSS [Mil89], CSP [Hoa85] and the process algebra ACP [BK85, BW90]. The distinctions among them arise from issues such as the process constructions considered (i.e. the language of processes), the methods used for giving meaning to process terms (i.e. the semantics), and the methods to reason about process behavior (e.g. process equivalences or process logics). Some other issues addressed in the theory of these calculi are their expressive power, and analysis of their behavioral equivalences.

2.2 Concurrent Constraint Programming

Concurrent Constraint Programming (`ccp`) [SRP91, Sar93] is a simple but powerful formalism to model concurrent systems. This model is based on the shared-variables communication model and a few primitives taking root in logic. A fundamental feature in `ccp` model is the specification of concurrent systems in terms of constraints. A constraint (e.g. $x + y > 42$) is a first-order formula representing partial information about certain variables. The `ccp` model is parameterized in a constraint system which provides a signature from which constraints can be built and an entailment relation (\models) specifying interdependencies between constraints (e.g. $x + y > 42 \models x + y > 0$).

During computation, the current state of the system is specified by a set of constraints called the *store*. Conceptually, the store in *ccp* is the medium through which agents interact with each other. The *ccp* processes can update the state of the system by *telling* information to the store (i.e. adding constraints). This is represented as the (logical) conjunction of the constraint being added and the store representing the previous state. Furthermore, processes can synchronize by *asking* information to the store (i.e. determining whether a given constraint can be inferred from the store). Ask processes block until there is enough information in the store to entail (i.e. answer positively) their query. A *ccp* computation terminates whenever it reaches a point called *quiescent point*, in which no more new information can be added to the store. The final store, also called *quiescent store* (i.e. the store at the quiescent point), is the output of the computation.

In the spirit of process calculi, the language of processes in the *ccp* model is given with a reduced number of primitive operators or combinators. A typical *ccp* process language features the following operators:

- A *tell* operator adding a constraint to the store.
- An *ask* operator querying if a constraint can be deduced from the store.
- *Parallel Composition* combining processes concurrently.
- A *hiding* operator (or *locality*) introducing local variables that delimit the interface through which a process can interact with others.
- *Recursion* defining infinite behavior.

2.2.1 Reactive Systems and Timed CCP

Reactive systems [BG92] are those that react continuously with their environment (e.g. a controller or a signal-processing system). These systems typically operate in a cyclic fashion; in each cycle they receive an input (stimulus) from the environment, compute on this input, and then return the corresponding output to the environment.

Languages such as Esterel [BG92], Lustre [HCRP91] and Signal [BG91] have been proposed in the literature for programming reactive systems. Those languages are based on the hypothesis of *Perfect Synchrony*: program combinators are determinate primitives that respond instantaneously to input signals.

The timed *ccp* calculus (*tcc*) [SJG94a, SJG94b] is an extension of *ccp* aimed at programming and modeling timed, reactive systems. In *tcc* the notion of time is conceptually divided into *time intervals* (or *time units*). In each time interval, a deterministic *ccp* process gets as input a constraint from the environment, it executes with this input as the initial *store*, and when it reaches its quiescent point, it outputs the resulting store to the environment. The quiescent point determines a residual process which is then executed in the next time unit. The resulting store is not automatically transferred to the next time unit.

In particular, the *tcc* model extends the standard *ccp* with *delay* and *time-out* operations. The delay operation forces the execution of a process to be postponed to the next time interval. The

time-out operation waits during the current time interval for a given piece of information to be present and if it is not, triggers a process in the next time interval.

We postpone the presentation of the syntax and the operational semantics of `tcc` to Chapter 3.

2.3 Transition Systems

Transitions systems [BK08] are often used to reason about the behavior of a system. They are basically directed graphs where nodes represent the set of possible states (the *state space*), and edges model how the system can evolve from one state into the other (the *transition relation* between states). A state describe some information about a system at a certain moment of its behavior. Action names represent communication mechanisms between processes, and atomic propositions intuitively express simple known facts about the states of the system and formalize temporal characteristics. In the following, we assume that time is discrete; in other words, a behavior will consist of an enumerable number of states.

Definition 2.1 (Transition System). A *transition system* TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- S is a set of *states*,
- Act is a set of *actions*,
- $\rightarrow \subseteq S \times Act \times S$ is a *transition relation*,
- $I \subseteq S$ is a set of initial states,
- AP is a set of *atomic propositions*, and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

The transition relation \rightarrow denotes possible state changes; if $(s, a, s') \in \rightarrow$ we say that the system can move from state s to s' performing action a . As a more compact notation, we usually write $s \xrightarrow{a} s'$.

The labeling function L relates a set of atomic propositions to any state s . $L(s)$ intuitively stands for exactly those atomic propositions $a \in AP$ which are satisfied by state s .

Next we present the definition of some notions that are important in transition systems.

Definition 2.2 (Direct Predecessors and Successors). Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. For $s \in S$ and $\alpha \in Act$, the set of direct α -*successors* of s is defined as:

$$Post(s, \alpha) = \{s' \in S \mid s \rightarrow s'\}, \quad Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$$

The set of α -*predecessors* of s is defined by:

$$Pre(s, \alpha) = \{s' \in S \mid s' \rightarrow s\}, \quad Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha)$$

Definition 2.3 (Terminal State). State s in transition system TS is called *terminal* if and only if $Post(s) = \emptyset$.

Intuitively, terminal states of a transition system TS are states without any outgoing transitions. Once the system reaches a terminal state, the complete system comes to a halt.

Definition 2.4 (Execution Fragment). Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *finite* execution fragment ρ of TS is an alternating sequence of states and actions ending with a state

$$\rho = s_0\alpha_1s_1\alpha_2\dots\alpha_ns_n \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i < n,$$

where $n \geq 0$. We refer to n as the length of the execution fragment ρ . An *infinite* execution fragment ρ of TS is an infinite, alternating sequences of states and actions:

$$\rho = s_0\alpha_1s_1\alpha_2\dots \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } i \geq 0$$

Definition 2.5 (Maximal and Initial Execution Fragment). A *maximal* execution fragment is either a finite execution fragment that ends in a terminal state, or an infinite execution fragment. An execution fragment is called *initial* if it starts in an initial state (i.e. if $s_0 \in I$).

Definition 2.6 (Execution). An *execution* of transition system TS is an initial, maximal execution fragment.

Definition 2.7 (Reachable States). Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A state $s \in S$ is called *reachable* in TS if there exists an initial, finite execution fragment

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n = s$$

Let us now introduce a well-known class of transition system relevant for the Chapter 4.

2.3.1 Kripke Structure

A *Kripke Structure* is used to capture the behavior of a system. This structure consists of a set of states, a set of transitions between states and a function that labels each state with a set of properties that are true in that state. Paths in a Kripke Structure model computations of the system.

Formally, a Kripke Structure is defined as follows:

Definition 2.8 (Kripke Structure [CGP99]). Let AP be a set of *atomic propositions*. A *Kripke Structure* M over AP is a 4-tuple $M = (S, S_0, R, L)$ where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.

4. $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

A *path* in the structure M from a state s is an infinite sequence of states $\pi = s_0s_1s_2\dots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

2.4 Linear-Time Temporal Logic

Temporal logic [HR00] is a formalism which provides a very intuitive and precise notation for specifying and verifying properties of reactive systems. This logic extends propositional or predicate logic by adding modalities that permit to represent the infinite behavior of a reactive system. Temporal logics were introduced into computer science by Pnueli [Pnu77] and thereafter proven to be a good basis for the specification as well as (automatic and machine-assisted) reasoning about concurrent systems.

The underlying nature of time in temporal logic can be either *linear* or *branching*. In the linear view, at each moment in time there is a single successor moment, whereas in the branching view it has a branching, tree-like structure, where time may split into alternative courses.

In Chapter 5, we shall present a linear-temporal logic which is essential for our model checking algorithm. For this reason, in this section we will focus our attention on *Linear Temporal Logic* (LTL), a propositional temporal logic (i.e. extension of propositional logic by temporal modalities) that is based on a linear-time perspective. In following, we recall the syntax and semantics of LTL.

2.4.1 Syntax

This subsection describes the syntactic rules according to which a formula of LTL can be constructed. The basic elements of a LTL-formula are atomic propositions (state labels $a \in AP$), the Boolean connectors like conjunction \wedge , and negation \neg , and two basic temporal modalities \circ (next) and \mathcal{U} (until). In the following, we present an intuitive explanation of the LTL operators.

The operator \circ is an unary operator. It is used to specify properties not for the current state but for the next state of a path. This is depicted in the Figure 2.1.

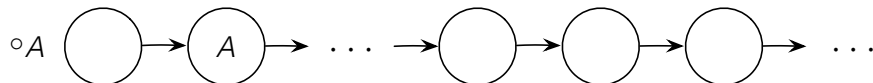
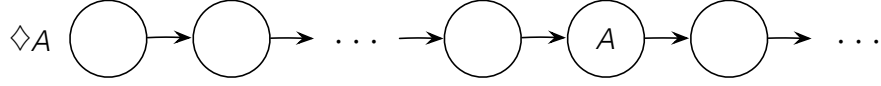
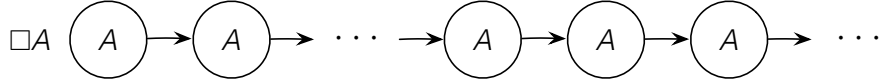


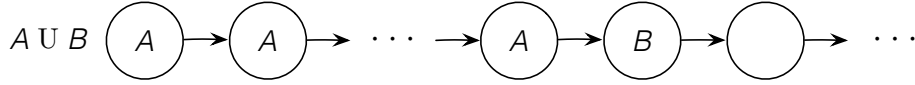
Figure 2.1: The \circ (next time) operator

The operator \diamond is an unary operator. It is used to specify properties for some future state, further down the execution path. The operator does not specify exactly which successor will have that property. It only promises that eventually something will happen. This can be seen visually in Figure 2.2.

The operator \square is an unary operator. It is used to specify properties for the current state and all its successors. Figure 2.3 shows this operator.

Figure 2.2: The \diamond (sometimes in the future) operatorFigure 2.3: The \square (always in the future) operator

The operator \mathcal{U} is a binary operator. For example the formula $\varphi\mathcal{U}\psi$ states that φ will be true until ψ . That is ψ will be true at some time in the future but until that time φ will be true. Figure 2.4 shows this behavior visually.

Figure 2.4: The \mathcal{U} (until) operator

Next we define formally the syntax of LTL.

Definition 2.9 (Syntax of LTL [BK08]). LTL formulae over the AP of atomic proposition are formed according to the following grammar:

$$\varphi ::= \mathbf{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \circ\varphi \mid \varphi_1\mathcal{U}\varphi_2$$

where $a \in AP$

Using the Boolean connectives \wedge and \neg , the full power of propositional logic is obtained. Other Boolean connectives such as disjunction \vee , implication \rightarrow , and equivalence \leftrightarrow can be derived as follows:

$$\begin{aligned} \varphi_1 \vee \varphi_2 &\stackrel{\text{def}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \varphi_1 \rightarrow \varphi_2 &\stackrel{\text{def}}{=} \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 \leftrightarrow \varphi_2 &\stackrel{\text{def}}{=} (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \end{aligned}$$

The until operator allows to derive the temporal modalities \diamond (“eventually”, sometime in the future) and \square (“always”, from now on forever) as follows:

$$\diamond\varphi \stackrel{\text{def}}{=} \mathbf{true}\mathcal{U}\varphi \qquad \square\varphi \stackrel{\text{def}}{=} \neg\diamond\neg\varphi$$

2.4.2 Semantics

LTL formulas stand for properties of paths (or in fact their trace). This means that a path can either fulfill an LTL-formula or not. The semantics of LTL formula φ is defined as a language $Words(\varphi)$ that contains all infinite words over the alphabet 2^{AP} that satisfy φ .

Definition 2.10 (Semantics of LTL (Interpretation over Words) [BK08]). Let φ be an LTL formula over AP . The LT property induced by φ is

$$Words(\varphi) = \{\sigma \in (2^{AP})^\omega \mid \sigma \models \varphi\}$$

where the satisfaction relation $\models \subseteq (2^{AP})^\omega \times \text{LTL}$ is the smallest relation with the properties in Figure 2.5.

Here, for $\sigma = A_0A_1A_2 \dots \in (2^{AP})^\omega$, $\sigma[j \dots] = A_jA_{j+1}A_{j+2} \dots$ is the suffix of σ starting in the j -th symbol of A_j .

$\sigma \models \mathbf{true}$	
$\sigma \models a$	iff $a \in A_0$ (i.e. $A_0 \models a$)
$\sigma \models \varphi_1 \wedge \varphi_2$	iff $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$
$\sigma \models \neg\varphi$	iff $\sigma \not\models \varphi$
$\sigma \models \circ\varphi$	iff $\sigma[1 \dots] = A_1A_2A_3 \dots \models \varphi$
$\sigma \models \varphi_1 \mathcal{U} \varphi_2$	iff $\exists j \geq 0. \sigma[j \dots] \models \varphi_2$ and $\sigma[i \dots] \models \varphi_1$, for all $0 \leq i < j$
$\sigma \models \diamond\varphi$	iff $\exists j \geq 0. \sigma[j \dots] \models \varphi$
$\sigma \models \square\varphi$	iff $\forall j \geq 0. \sigma[j \dots] \models \varphi$

Figure 2.5: LTL semantics for infinite words over 2^{AP}

2.5 Model Checking

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model [BK08]. It was first introduced in [CE82] and [QS82]. In essence, this technique consists in an exhaustive analysis of the state-space of the system to determine if some specification is true or false.

The main drawback of the model checking technique is that the state space of a system can be huge and it is difficult (or impossible) to build its model (*the state space explosion problem*). On the other hand, this technique has two major advantages: it is fully-automatic and its application requires no user supervision or expertise in mathematical disciplines (as opposed to completely deductive techniques) and when a state violates the property, the model checker provides a counterexample that indicates how the model could reach the undesired state.

In following we describe the different phases of model checking [BK08]. The required input to model checking are a model of the system under consideration and a formal specification of the property to be verified.

1. *Modeling* phase.
 - Model the system under consideration using the model description language that can be handled by the model checker. Models of systems describe the behavior of systems in a formal way. They are mostly expressed using state transition graphs.
2. *Specification* phase.
 - Describe the property to be checked using a property specification language. For this purpose temporal logic is used.
3. *Running* phase.
 - Run the model checker to check the validity of the property in all states of the system model.
4. *Analysis* phase.
 - There are three possible outcomes: the specified property is either valid in the given model or not, or the model turns out to be too large to fit within the physical limits of the computer memory.

In Figure 2.6, we present the phases of the model checking technique in a schematic view. We shall address the above mentioned phases in Chapter 4, 5 and 6, respectively of this document.

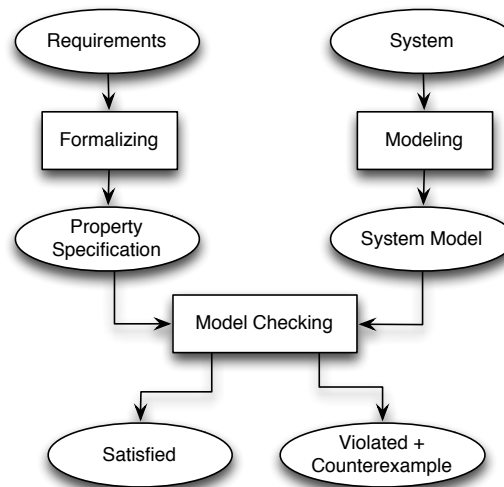


Figure 2.6: Schematic view of the model-checking approach [BK08]

The `tcc` Language

Concurrent Constraint Programming (`ccp`) [SRP91, Sar93] is a powerful paradigm for concurrency. The fundamental issue of the `ccp` model is the specification of concurrent systems in terms of constraints. A constraint represents partial information about the variables of the system. A temporal extension of `ccp` is the *timed concurrent constraint programming* (`tcc`) [SJG94a, SJG94b]. In particular, `tcc` extends the deterministic fragment of the `ccp` paradigm with agents that are able to model temporal behavior and also notions such as *timeout* and *preemption*. These operations are fundamental for programming reactive systems. In this chapter, we recall the syntax of the `tcc` calculus as well as its operational semantics.

We shall begin by introducing the notion of a constraint system which is fundamental to `ccp` based calculi. We then present the formal syntax of `tcc` and its operational semantics. We shall describe the basic agents of `tcc` in an intuitive way. Finally, we shall specify a simple program using `tcc`.

3.1 Constraint System

Concurrent Constraint Programming (`ccp`) based calculi are parametric in a *constraint system* [Sar93] which specifies the basic constraints that agents can tell or ask during execution. In this section we present the definition of constraint system.

A constraint represents a piece of partial information upon which processes may act. A constraint system then provides a signature from which constraints can be built. Furthermore, the constraint system provides an *entailment* relation (\models) specifying interdependencies between constraints. Intuitively, $c \models d$ means that the information d can be deduced from the information represented by c (e.g. $x > 60 \models x > 42$). We next define the notion of constraint system based on First-Order Logic as in [Smo94, NPV02].

Definition 3.1 (Constraint System). A *constraint system* is a pair (Σ, Δ) where Σ is a signature specifying constants, functions and predicate symbols, and Δ is a consistent first-order theory over Σ (i.e. a set of first-order sentences over Σ having at least one model).

Given a constraint system (Σ, Δ) , let \mathcal{L} be the underlying first-order language with a countable set of variables x, y, \dots , and logic symbols $\neg, \wedge, \vee, \Rightarrow, \exists, \forall, \mathbf{true}$ and \mathbf{false} which denote logical negation, conjunction, disjunction, implication, existential and universal quantification, and the always true and false predicates, respectively. *Constraints*, denoted by c, d, \dots , are first-order formulas over \mathcal{L} . We say that c *entails* d in Δ , written $c \models_{\Delta} d$ iff the formula $c \Rightarrow d$ is true in

all models of Δ . We write \models instead of \models_{Δ} when Δ is unimportant or can be inferred from the context. We say that c is equivalent to d , written $c \equiv d$, iff $c \models_{\Delta} d$ and $d \models_{\Delta} c$.

Henceforth we shall use the following notation.

Notation 3.1 (Constraints and Equivalence). Henceforth, \mathcal{C} denotes the set of constraints modulo \equiv in the underlying constraint system. So, we write $c = d$ iff c and d are in the same (\equiv) class. Furthermore, whenever we write expressions such as $c = (x = y)$ we mean that c is (equivalent to) the constraint $x = y$.

3.2 Syntax

In the `ccp` model, the information in the store evolves *monotonically* (i.e. once a constraint is added it cannot be removed). In `tcc`, time is conceptually divide into *time intervals* (or *time units*). In a particular time interval, a `ccp` agent P gets an input c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines a residual process Q which is then executed in the next time unit. The resulting store d is *not automatically transferred* to the next time unit.

Following the notation in [SJG94a, SJG94b] the syntax of `tcc` is presented in Figure 3.1.

(Agents)	A	$::=$	c	(Tell)
			now c then A	(Timed Positive Ask)
			now c else A	(Timed Negative Ask)
			next A	(Unit Delay)
			abort	(Abort)
			skip	(Skip)
			$A \parallel A$	(Parallel Composition)
			$\exists x(A)$	(Hiding)
			g	(Procedure Call)
(Procedure Calls)	g	$::=$	$p(t_1, \dots, t_n)$	
(Declarations)	D	$::=$	$g :: A$	(Definition)
			$D.D$	(Conjunction)
(Programs)	P	$::=$	$D.A$	

Figure 3.1: Syntax of `tcc`

In the following description we present an intuitive definition of the `tcc` agents.

- **CCP constructs:** These agents do not cause extension over time.

Tell. This agent adds c to the store in the current time unit.

Skip. This agent does nothing thus representing inaction.

Timed positive ask. This agent checks if c can be deduced from the current store. If so, it behaves as A . In other case, it remains blocked until the store contains at least as much information as c .

Parallel composition. This agent denotes two processes running concurrently during the current time unit.

Hiding. This agent behaves like A , except that all the information on the variables x produced by A can only be seen by A and the information on the global variable x produced by other processes cannot be seen by A .

- **Timed Constructs:** These constructs cause extension over time.

Timed negative ask. This agent executes A in the next time unit if and only if, on the quiescence of the current time unit, c is *not* entailed by the store.

Unit delay. This agent executes A in the next time unit.

Abort. This agent terminates the execution of all processes in the next time unit.

Using the basic constructs presented in Figure 3.1 we can define other derived constructs. Such constructs make easier to the user to use the language since makes more intuitive the specification of programs. For example, the `always A` agent behaves like A at every time instant. This agent is defined as follows [SJG94b]:

$$\text{always } A = A \parallel \text{next always } A$$

3.3 Operational Semantics

The operational semantics of `tcc` considers *transitions* over *configurations*. A configuration is defined as a multiset $?$ of agents. The store in a configuration $?$ is represented by $\sigma(?)$ and it denotes the sub-multiset of tokens in $?$. Moreover, the semantics is given in terms of the binary transition relations \rightarrow and \rightsquigarrow . The relation \rightarrow represents transitions within a time instant (*internal transition*), and the relation \rightsquigarrow represents a transition from one time instant to the next (*temporal transition*).

To ensure that computation in each time-step is lexically bounded (i.e. bounded by the size of the program) the recursion variable occurs within the scope of an **else** or a **next**.

The operational semantics given in [SJG94a, SJG94b] is presented in Figure 3.2.

3.4 Program Example

In this section we model a system that controls the behavior of an electronic door. The door opens every time there is a person in front of it, otherwise it keeps closed. We assume that the door opens when it receives the signal `x=2`, and closes when it receives the signal `x=1`. Moreover, we assume that the system has a sensor which sends the signal `in=true` when there is a person in front of the door. In Figure 3.3, we show a simple `tcc` specification of the above system.

Axioms for \rightarrow . The binary relation \rightarrow on configurations is the least relation satisfying the rules :

$$\begin{array}{lcl}
(? , \text{skip}) & \rightarrow & ? \\
(? , \text{abort}) & \rightarrow & \text{abort} \\
(? , A \parallel B) & \rightarrow & (? , A , B) \\
(? , \exists x(A)) & \rightarrow & (? , A[y/x]) \quad (y \text{ not free in } ?) \\
(? , p(t_1, \dots, t_n)) & \rightarrow & (? , A[x_1 \mapsto t_1, \dots, x_n \mapsto t_n]) \\
\frac{\sigma(?) \vdash c}{(? , \text{now } c \text{ then } A) \rightarrow (? , A)} & & \\
\frac{\sigma(?) \vdash c}{(? , \text{now } c \text{ else } B) \rightarrow ?} & &
\end{array}$$

Axioms for \rightsquigarrow . The binary relation \rightsquigarrow is the least relation satisfying the single rule :

$$\frac{\Delta, \{\text{now } c_i \text{ else } A_i \mid i < n\} \rightsquigarrow}{\Delta, \{\text{now } c_i \text{ else } A_i \mid i < n\}, \{\text{next } B_j \mid j < m\} \rightsquigarrow \{A_i \mid i < n\}, \{B_j \mid j < m\}}$$

Figure 3.2: Operational semantics for tcc language

```

p() ::
  now (in=true) then next(tell(x=2)) ||
  now (in=true) else tell(x=1) || next(p())

```

Figure 3.3: Example of a tcc program

3.5 Summary

In this chapter we described the syntax and operational semantics of tcc. We introduced the notion of constraint system to which tcc calculus is parametric. We also showed with an example how to specify a system using the tcc agents.

The tcc calculus [SJG94a, SJG94b] is an extension of the ccp model [SRP91, Sar93] which extends the deterministic fragment of the ccp paradigm with agents that are able to model temporal behavior and also notions such as timeout and preemption.

A Model for `tcc` Programs

In the previous chapter we presented the *syntax* and the *operational semantics* of `tcc` and we showed an example of how to specify a system with this formalism. The first phase of the model checking technique is to construct a model that faithfully represents the behavior of the specified system.

In this chapter, we show our approach to construct the model of the system from a `tcc` specification. That is to say, we take a program written in `tcc`, and we construct the model in a systematic way. This approach is based on the ideas developed by Falaschi and Villanueva [FV06] to model `tccp` programs. Nevertheless, the structure defined here is quite different from the structure presented in [FV06].

We shall start by defining a labeling process in order to identify in which point of the execution of the program we are, and also to determine if an agent can be executed or not during the computation. We then define a structure called *tcc Structure* which allows us to model the behavior of a system specified in `tcc`. Furthermore, we describe how to construct the `tcc` Structure from a labeled specification and we show as example the model of the `tcc` program presented in the previous chapter. Finally, we show how to reduce the number of states generated in the construction of the system model.

4.1 Program Labeling

The *labeling* process consists in assigning a different label to each occurrence of an agent in the program. The authors in [FV06] adapted the idea introduced by Manna and Pnueli in [MP95] to their framework. Labels fulfill the role of providing an unique identification for agents in order to know in which point of the execution of the program we are during the construction of the model. The presence or absence of a label determines if the associated agent can be executed or not during the computation.

In the following definition we specify how to transform a `tcc` specification to its labeled version.

Definition 4.1. Let P be a `tcc` specification, the labeled version P_l of P is defined as follows. The subindex $k \in \mathbb{N}$ corresponds to the number of labels introduced up to a given point. When the labeling process starts, $k = 0$ and each time that we introduce a new label, k is incremented by one.

- If $P = \text{tell } c$ then $P_l = \{l_{\text{tell}_k}\} \text{tell } c$.
- If $P = \text{now } c \text{ then } A$ then $P_l = \{l_{\text{nowp}_k}\} \text{now } c \text{ then } A_l$.
- If $P = \text{now } c \text{ else } A$ then $P_l = \{l_{\text{nown}_k}\} \text{now } c \text{ else } A_l$.

- If $P = \text{next } A$ then $P_l = \{l_{\text{next}_k}\} \text{next } A_l$.
- If $P = \text{skip}$ then $P_l = \{l_{\text{skip}_k}\} \text{skip}$.
- If $P = A \parallel B$ then $P_l = \{l_{\parallel_k}\} (A_l \parallel B_l)$.
- If $P = p(t_1, \dots, t_n)$ then $P_l = \{l_{p_k}\} p(t_1, \dots, t_n)$.

The labeling of a declaration D of the form $p(x) : - A$ is defined as $\{l_{p_k}\} p(x) : - A_l$, called D_l . Finally, the labeled version of a program of the form $D.A$ is $D_l.A_l$.

Fundamentally, the labeling process consists in exploring the tcc specification, and each time that we find an agent we introduce a new label. In Figure 4.1, we show the labeled version of the tcc program in Figure 3.3. Note that the structure of the program has not changed, only labels have been added to each occurrence of an agent.

```

{lp0}p() ::
  {l||1}( {lnowp2}now (in=true) then {lnext3}next( {ltell4}tell(x=2)) ||
    {l||5}( {lnown6}now (in=true) else {ltell7}tell(x=1) ||
      {lnext8}next( {lp9}p() ) ) ) )

```

Figure 4.1: Example of a labeled tcc program

4.2 The tcc Structure

In this section we define a structure called *tcc Structure* that allows to model the behavior of a system specified in tcc. This structure is a variant of a *Kripke Structure* (see Definition 2.8). Intuitively, a Kripke Structure is a *finite* graph structure which could have many initial nodes and each node is always related to another one (or to itself). Additionally, each state has associated a set of atomic propositions which are true in such state.

The main difference between the above mentioned structures lies in the definition of state; the Kripke Structure adopts the classical notion of state whereas in the tcc Structure, a state consists of a conjunction of constraints and intuitively it can be seen as a set of classical states (i.e. a set of assignments).

Before formally defining the tcc Structure, we require the definition of some concepts. First, we need to define what is the set of propositions AP of atomic propositions.

Definition 4.2. The set AP of atomic propositions is defined as the set of elements in the constraint system \mathcal{C} .

In the rest of the document, we abuse of notation by using the term *constraint* as an equivalent concept to *atomic proposition*. Next we present the definition of state in the tcc Structure.

Definition 4.3 (tcc State). Let AP be the atomic propositions in the tcc syntax and L be the set of all labels generated during the labeling process described above. We define the set of states as $S \subseteq 2^{AP} \times 2^L$.

Now we define the notion of equivalent states. For this, we need the classical notion of renaming of variables. Let y_1, \dots, y_n be n distinct variables, the substitution $\{x_1/y_1, \dots, x_n/y_n\}$ is a *renaming* whenever the sets y_1, \dots, y_n and x_1, \dots, x_n are disjoint.

Definition 4.4 (Equivalent States). Given two tcc states s and s' , we say that two states are equivalent if

1. the set of labels of s and s' coincide, and
2. there exists a renaming γ of variables of the constraints in s which makes them syntactically identical to the set of constraints of s' .

Finally, we define the tcc Structure. We consider *internal* and *temporal* labels to identify the agents that can be executed on the same or the next time unit, respectively.

Definition 4.5 (tcc Structure). Let AP be a set of atomic propositions, we define a tcc Structure M over AP as a seven tuple $M = (S, S_0, T, R, C, L, LT)$ where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $T = \{i, t\}$ is the set of possible type of transitions. i denotes an internal transition while t denotes a temporal transition.
4. $R \subseteq S \times S \times T$ is a transition relation.
5. $C : S \rightarrow 2^{AP}$ is the function that returns the set of atomic propositions in a given state.
6. $L : S \rightarrow 2^L$ is the function that returns the set of internal labels in a given state.
7. $LT : S \rightarrow 2^L$ is the function that returns the set of temporal labels in a given state.

We assume that an *internal transition* in the graph represents a computation within the same time unit in the system, and a *temporal transition* represents an increment of one time unit. In Figure 4.2, we show the graphical representation of the transitions in our framework.

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 (*current store*). L labels a state with the set of labels associated to agents that must be executed within the same time instant (*internal labels*). LT labels a state with the set of labels associated to agents that must be executed in the following time instant (*temporal labels*). For better readability we graphically represent a tcc state with three spaces: *current store*, *internal labels* and *temporal labels* (see Figure 4.2).

Notice that the differences w.r.t a Kripke Structure (see Definition 2.8) are the definition of state (see Definition 4.3) and the three labeling functions C , L and LT which replace the labeling function L of the Kripke Structure.

When two states s and s' are related by $R(s, s', T)$, it means that is possible to reach the state s' from state s by executing the agents associated to the labels in $L(s)$ ($T = i$, internal transition) or $LT(s)$ ($T = t$, temporal transition) with the store $C(s)$ deriving as a result (by applying the renaming γ) the store $C(s')$ and the point of execution $L(s')$ and $LT(s')$.

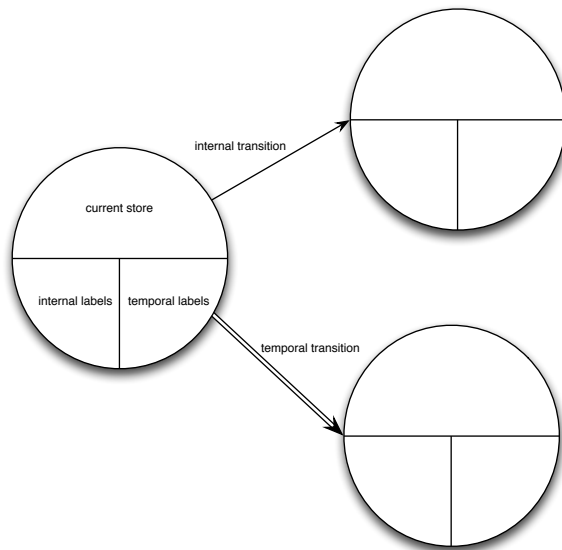


Figure 4.2: Representation of a tcc state

4.3 Construction of the Model

In this section we explain in an intuitive way how to construct a tcc Structure from a tcc specification. A specification is composed by a set of declarations. Then, for each different declaration we construct a tcc Structure that is labeled with a unique name and it is used when a procedure call refers to the body of such declaration. Essentially, transitions are described according to the operational rules of each tcc process.

Each state is composed of labels associated with agents that can be executed in a step of the construction process. Each label can be active or disabled. A label is active when the conditions to execute the agent associated are satisfied, and disabled when the agent associated cannot be executed in that moment because the store does not entail the necessary conditions. The labels associated with temporal agents (i.e. `next`, `now c else A`, etc.) cannot be executed before all the labels associated with normal agents (i.e. agents that do not cause extension over time) are executed. This is because only after that we can be sure that no more information can be produced in the present time instant.

Our procedure consists in locating an active label and perform the actions associated with such agent. The process is performed while there are active labels. When we reach a state where there are no active labels (*quiescent point*) we have to pass to the next time unit, and then we continue with the procedure. We represent this passage in our graph as follows:

1. Introduce a new node s' related with s by a temporal transition. The state s is a state where there is no active labels.
2. We introduce the temporal labels of s in the labels of s' .

3. The store and the temporal labels of s' are empty.

As mentioned above, each time an agent is analyzed some actions are executed. In the following description we show the actions performed by each agent in order to construct the graph structure. The created nodes in the following steps are connected by an internal transition with the predecessor node. We use the notation $\sim c$ to denote that the current store does not entail the constraint c .

Tell $S \equiv \{l_{\text{tell}_k}\} \text{ tell } c$. The new information c is added to the store in the current time. We translate this behavior to our graph structure as follows:

1. Add a new node s' related with the node s from which the agent is execute.
2. The store of s' is defined as the store of s plus the constraint c (i.e. $C(s') = C(s) \wedge c$).
3. The internal labels of s' are obtained from those of s by removing $\{l_{\text{tell}_k}\}$ (i.e. $L(s') = L(s) \setminus \{l_{\text{tell}_k}\}$).
4. The temporal labels of s' are the same as s (i.e. $LT(s') = LT(s)$).

Parallel $S \equiv \{l_{\parallel k}\} (A_l \parallel B_l)$. The agents A and B are executed in parallel. We translate this behavior to our graph structure as follows:

1. Introduce a new node s' related with the node s from which the agent is execute.
2. The internal labels of s' are obtained from those of s by adding A_l and B_l , and removing $\{l_{\parallel k}\}$. Notice that this corresponds to a concurrent semantics rather than an interleaving interpretation of the parallel operator.
3. The store and the temporal labels of s' are the same as s .

Timed Positive Ask $S \equiv \{l_{\text{nowp}_k}\} \text{ now } c \text{ then } A_l$. If the current store entails c then the agent A is executed or does nothing otherwise. Next we describe how to translate this agent to our graph.

1. Add two new nodes s'_1, s'_2 related with the node s from which the agent is execute. This branch corresponds to the two possible behaviors.
2. The store of s'_1 is defined as the union of the store of s and the constraint c .
3. The store of s'_2 is defined as the store of s plus the absence of the constraint c (i.e. $C(s'_2) = C(s) \wedge \sim c$).
4. The internal labels of s'_1 are obtained from those of s by adding A_l and removing $\{l_{\text{nowp}_k}\}$.
5. The internal labels of s'_2 are the same as s .
6. The temporal labels of s'_1 and s'_2 are the same as s .

Skip $S \equiv \{l_{\text{skip}_k}\} \text{ skip}$. This agent does nothing at every time instant. We translate this agent to our graph as follows:

1. Construct a new node s' related with the node s from which the agent is executed.
2. The internal labels of s' are obtained from those of s by removing $\{l_{\text{skip}_k}\}$.

3. The store and the temporal labels of s' are the same as s .

Procedure Call $S \equiv \{l_{p_k}\} p(t_1, \dots, t_n)$. This operator refers to another procedure which have different labels and variables. To translate this agent to our graph we create a new node and the label associated to the first agent of p is added to the internal labels of the node.

Timed Negative Ask $S \equiv \{l_{\text{now}_k}\} \text{now } c \text{ else } A_l$. If on the quiescent point the store does not entail c then the agent A is executed in the next time instant, or does nothing otherwise. We translate this behavior to our graph as follows:

1. Introduce two new nodes s'_1, s'_2 related with the node s from which the agent is execute. This branch corresponds to the two possible behaviors of the agent.
2. The store of s'_1 is defined as the union of the store of s and the constraint c .
3. The store of s'_2 is defined as the store of s plus the absence of the constraint c .
4. The internal labels of s'_1 and s'_2 are the same as s by removing $\{l_{\text{now}_k}\}$.
5. The temporal labels of s'_1 are the same as s .
6. The temporal labels of s'_2 are obtained from those of s by adding A_l .

Unit Delay $S \equiv \{l_{\text{next}_k}\} \text{next } A_l$. The agent A is executed in the next time instant. We translate this agent to our graph as follows:

1. Construct a new node s' related with the node s from which the agent is execute.
2. The internal labels of s' are the same as s but removing $\{l_{\text{next}_k}\}$.
3. The temporal labels of s' are obtained from those of s by adding A_l .
4. The store of s' is the same as s .

When we generate a new node we must check if there are two equivalent states (see Definition 4.4). If we find a node s_2 equivalent to the new node created s_1 , then we must relate the predecessor of s_1 with the node s_2 , and delete the node s_1 . Moreover, the construction following this branch will terminate. The previous step allows to avoid generating unnecessary states.

4.4 Simplification of the Model

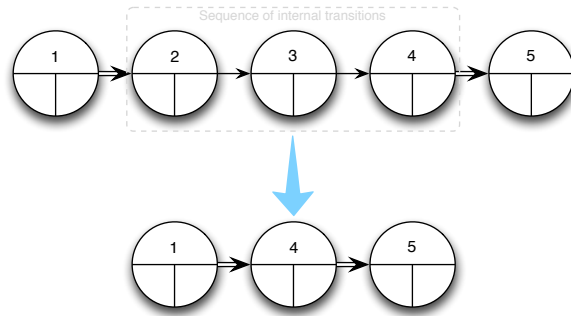
This section is devoted to describing the simplification process of the `tcc` Structure. Our approach to construct the model generates a large number of states due to the operational semantics of `tcc`. Therefore, a reduction of states is necessary to decrease the state explosion problem of the model checking technique.

In Chapter 3, we presented the operational semantics of `tcc`. This semantics is defined in terms of internal and temporal transitions. The internal transitions describe evolutions within a time unit, and thus they are regarded as being *unobservable*. On the other hand, the temporal transitions describe evolutions across the times units, and thus they are regarded as being *observable* [NV03]. A temporal transition is obtained by performing a sequence of internal transitions until no further

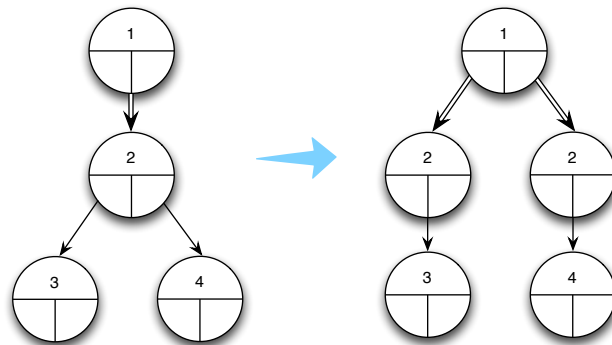
internal evolution is possible. In this point (*quiescent point*) no more information can be added to the store. This store is defined as the output of the computation. Therefore, we are interested only in maintaining the nodes where there is no active labels, and also the temporal transitions.

Hence, we simplify the **tcc** Structure compacting nodes which contain active labels. This nodes generate internal transitions, thus the final structure will have only temporal transitions. We must ensure that the new **tcc** Structure models the same behavior as the original. Taking the previous observations into account, we describe the procedure to simplify the structure as follows:

1. If we find a sequence of internal transitions without branching, then we collapse all nodes and we keep the last node in the sequence (see Figure 4.3(a)).
2. If we find a node s_1 related with a node s_2 which has a branching, then we divide the node s_2 into two identical nodes. Each node follows only one branch of s_2 , and s_1 is related with them (see Figure 4.3(b)).
3. We repeat these steps until no more simplifications can be done.



(a) Reduction of a sequence of internal transitions



(b) Reduction of a branching

Figure 4.3: Reduction rules

A more compelling application of our procedure of simplification is given in the next section where we show a complete example of the construction of a model.

4.5 Construction Example

In this section we shall illustrate the construction process of the tcc Structure and the simplification process. In Figure 4.4, we present the tcc Structure of the tcc program in Figure 3.3. For the sake of readability, we do not show the extension of the branch from the node 23.

Notice that the nodes 14 and 7 are equivalents, thus we delete node 14 and relate node 13 to node 7 through a temporal transition. We perform the same process with the nodes 18 and 22. To reduce the size of the graph, we execute all labels associated with the *unit delay* agent simultaneously (e.g. node 5). Moreover, the store in some nodes can satisfy the guard of the *time negative ask* agent, thus we construct only the corresponding branch and discard the other (e.g. node 4).

In Figure 4.5, we show the tcc Structure with the loops generated by the equivalent nodes. We use the simplification procedure described in the previous section in order to reduce the large number of nodes. First, we eliminate all the sequences of internal transitions (i.e. 1-3, 4-6, 7-10, 11-13, 15-17, 19-21, 22-25, 26-28 and 30-32). Figure 4.5 shows the graph after the first reduction. Then, we simplify the branches of the graph as shown in Figure 4.6. Finally, we delete the remaining sequences of internal transitions (i.e. 3-6, 10-13, 10-17, 3-21, 25-28, 25-32). Figure 4.7 shows the resulting graph after the simplification process. Notice the significant reduction in the number of states. Furthermore, we can observe that this graph maintains the behavior of the original graph and it contains only temporal transitions.

4.6 Summary

In this chapter we defined a structure which allows to model the behavior of a system specified in tcc. Furthermore, we defined in an intuitive way how to construct the model of the system from a tcc specification. Since a labeled version of the program is necessary for the construction of the model, we defined a labeling process.

We illustrate our approach by modeling a tcc specification, and showed that we can construct a finite graph from a reactive system that runs forever. This is possible thanks to the loops generated by some equivalent states. Moreover, we simplify the graph eliminating all the internal transitions in order to reduce the number of states to decrease the state explosion problem. Further simplifications can be done. For instance, if a given state has an active label associated with *timed positive ask* or *timed negative ask* agents, and the store only allows the agent to follow a branch. These reductions will be essential to our model checking algorithm defined in Chapter 6.

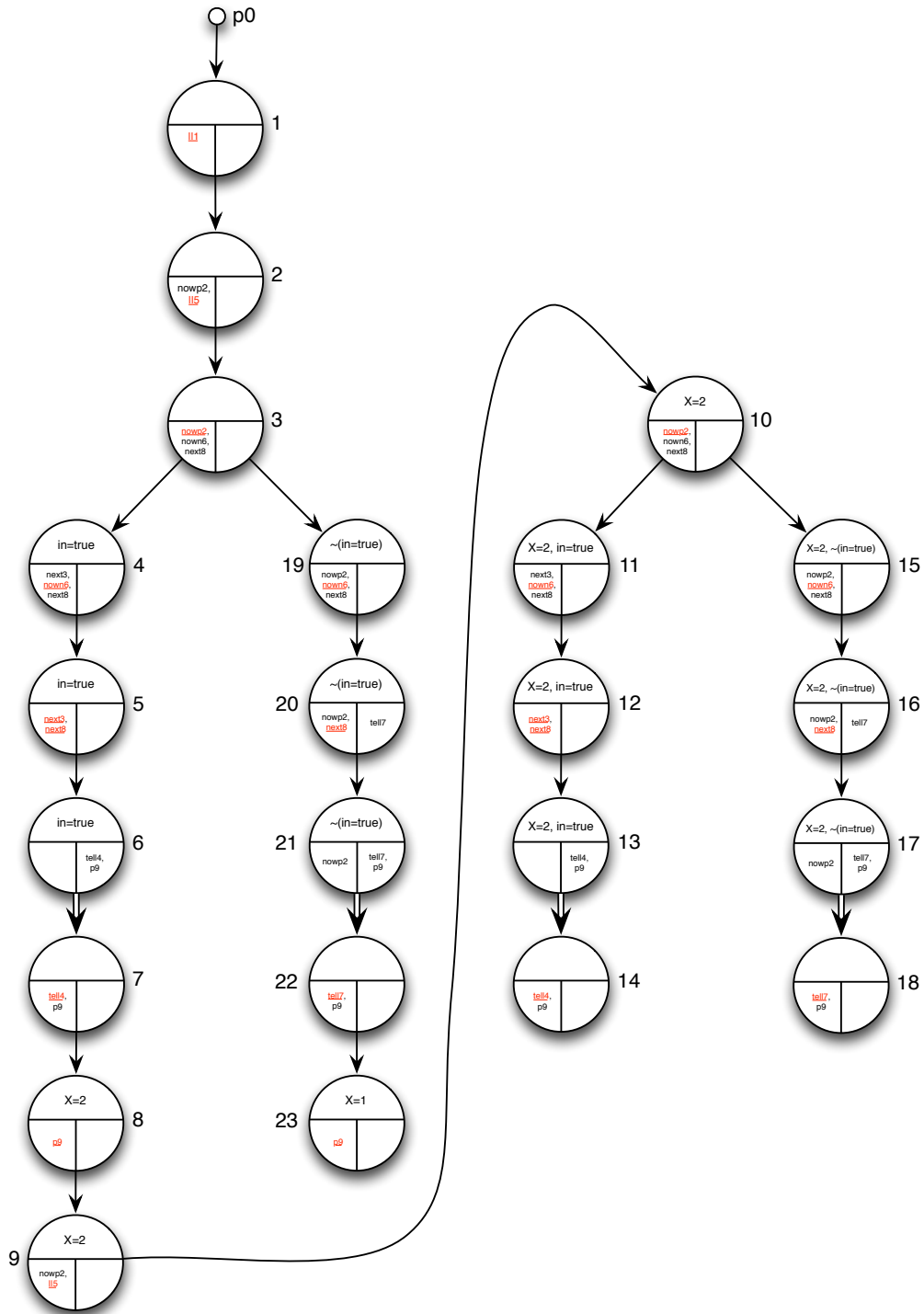


Figure 4.4: tcc Structure for the tcc program in Figure 3.3

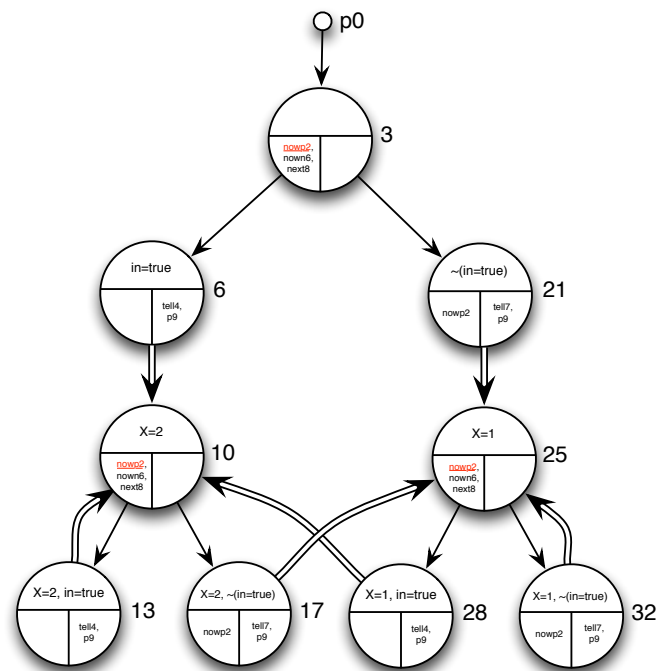


Figure 4.5: First reduction of the tcc Structure

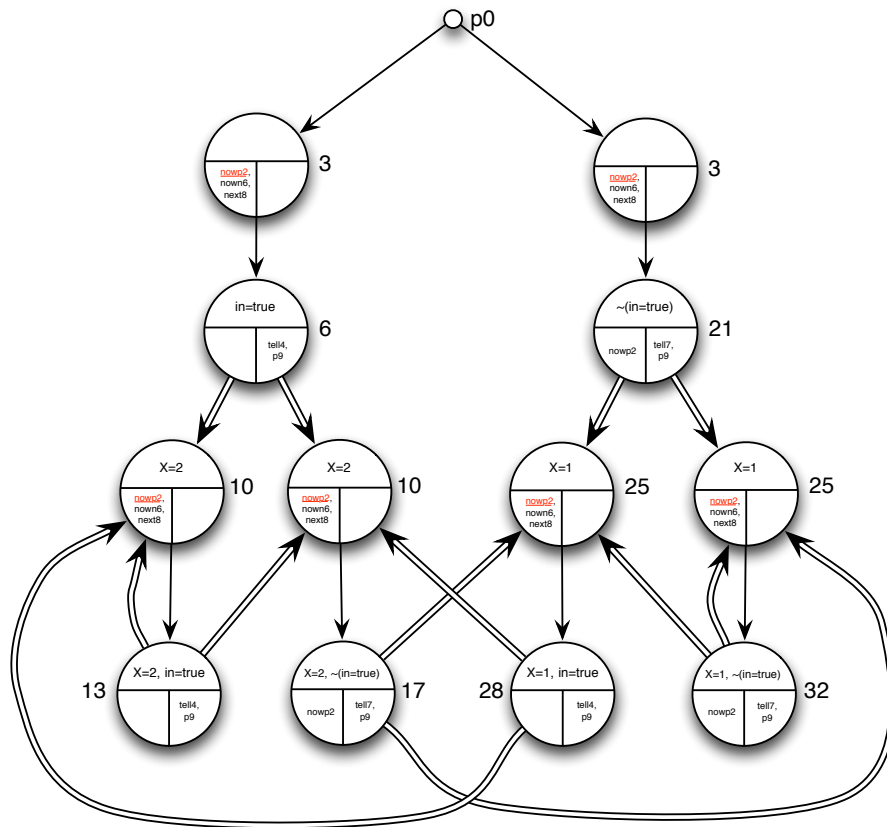


Figure 4.6: Second reduction of the tcc Structure

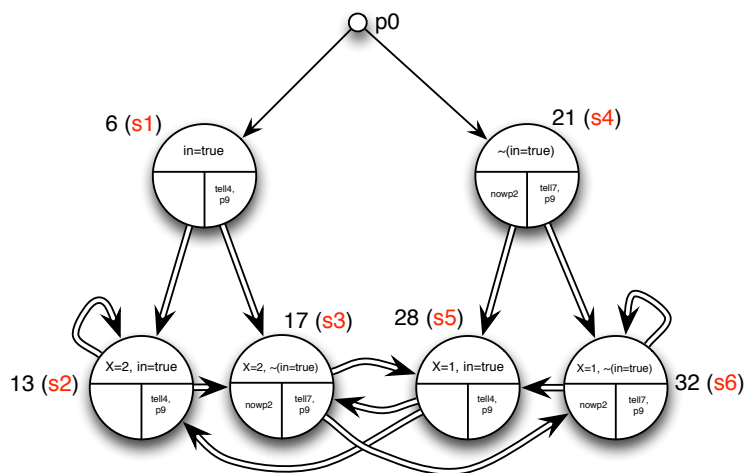


Figure 4.7: tcc Structure after the simplification process

A Logic for the Specification of Properties

In Chapter 3, we presented the `tcc` calculus. This formalism allows to specify complex reactive systems. Then, in Chapter 4 we presented an algorithm to construct a model of the system from the `tcc` specification. Now we need a temporal logic to reason about `tcc` systems and to express temporal properties of them. In this chapter we shall address the second phase of the model checking technique: to specify the property that we want to verify.

We shall start by defining the linear-time temporal logic that we shall use to express temporal properties of `tcc` systems. This logic is based on sequences of constraints. Finally, we specify a property of a `tcc` program using the temporal logic presented in this chapter.

5.1 The `ntcc` Logic

In this section we present the linear temporal logic which we use to specify properties of `tcc` systems in our framework. We start by defining the syntax of the logic and then we give a semantic interpretation of it.

In [Val05] Valencia presented a linear temporal logic (LTL) named **CLTL**. This logic expresses properties over sequences of constraints, and it is employed to reason about `ntcc` processes. The `ntcc` calculus is an extension of `tcc` which can represent timed concepts such as unit delays, unbounded finite delays, time-outs, pre-emption, synchrony and asynchrony [Val02]. We use this logic to reason about programs specified in `tcc` since it is a subcalculus of `ntcc`.

Before defining formally the above logic, we require the following notation on sequences of constraints.

Notation 5.1. Throughout this section \mathcal{C}^ω denotes the set of infinite (or ω) sequences of constraints in the underlying set of constraints \mathcal{C} . We use α, α', \dots to range over \mathcal{C}^ω .

5.1.1 Temporal Logic Syntax

The syntax of **CLTL** is given by the following definition.

Definition 5.1 (CLTL Syntax). The formulae $F, G, \dots \in \mathcal{F}$ are built from constraints $c \in \mathcal{C}$ in the underlying constraint system by

$$F, G, \dots := c \mid \mathbf{true} \mid \mathbf{false} \mid F \wedge G \mid F \vee G \mid \neg F \mid \exists_x F \mid \circ F \mid \square F \mid \diamond F$$

where c denotes a constraint representing a *state formula* c . The symbols **true**, **false**, \wedge , \vee , \neg and \exists represent linear-temporal logic true, false, conjunction, disjunction, negation and existential quantification. As clarified later, these symbols are not be confused with the symbols **true**, **false**, \wedge , \vee , \neg and \exists in the underlying constraint system. The symbol \circ , \square and \diamond denote the temporal operators *next*, *always* and *sometime*. We use $F \Rightarrow G$ as an abbreviation of $\neg F \vee G$.

5.1.2 Temporal Logic Semantics

The standard interpretation structures of linear temporal logic are infinite sequences of states [MP92]. Nevertheless, in **CLTL** the states are replaced by constraints, and it considers as interpretations the elements of \mathcal{C}^ω . Before defining the semantics of **CLTL**, let us introduce the notion of *x-variant*. But first we need the following notation.

Notation 5.2. Given a sequence $\alpha = c_1.c_2\dots$, we use $\exists_x\alpha$ to denote the sequence $\exists_x c_1.\exists_x c_2\dots$. We shall use $\alpha(i)$ to denote the i -th element of α .

Definition 5.2 (*x-variant* [MP92]). A constraint d is an *x-variant* of c iff $\exists_x c = \exists_x d$. Similarly α' is an *x-variant* of α iff $\exists_x \alpha = \exists_x \alpha'$

Intuitively, d and α' are *x-variants* of c and α , respectively, if they are the same except for the information about x . For example, $x = 1 \wedge y = 0$ is an *x-variant* of $x = 42 \wedge y = 0$.

We can now define the semantics of **CLTL**.

Definition 5.3 (CLTL Semantics). We say that $\alpha \in \mathcal{C}^\omega$ satisfies (or that it is a model of) F in **CLTL**, written $\alpha \models_{\text{CLTL}} F$, iff $\langle \alpha, 1 \rangle \models_{\text{CLTL}} F$, where:

$$\begin{array}{ll}
\langle \alpha, i \rangle \models_{\text{CLTL}} \mathbf{true} & \\
\langle \alpha, i \rangle \models_{\text{CLTL}} \mathbf{false} & \\
\langle \alpha, i \rangle \models_{\text{CLTL}} c & \text{iff } \alpha(i) \models c \\
\langle \alpha, i \rangle \models_{\text{CLTL}} \neg F & \text{iff } \langle \alpha, i \rangle \not\models_{\text{CLTL}} F \\
\langle \alpha, i \rangle \models_{\text{CLTL}} F \wedge G & \text{iff } \langle \alpha, i \rangle \models_{\text{CLTL}} F \text{ and } \langle \alpha, i \rangle \models_{\text{CLTL}} G \\
\langle \alpha, i \rangle \models_{\text{CLTL}} F \vee G & \text{iff } \langle \alpha, i \rangle \models_{\text{CLTL}} F \text{ or } \langle \alpha, i \rangle \models_{\text{CLTL}} G \\
\langle \alpha, i \rangle \models_{\text{CLTL}} \circ F & \text{iff } \langle \alpha, i + 1 \rangle \models_{\text{CLTL}} F \\
\langle \alpha, i \rangle \models_{\text{CLTL}} \square F & \text{iff for all } j \geq i \langle \alpha, j \rangle \models_{\text{CLTL}} F \\
\langle \alpha, i \rangle \models_{\text{CLTL}} \diamond F & \text{iff there is a } j \geq i \text{ s.t. } \langle \alpha, j \rangle \models_{\text{CLTL}} F \\
\langle \alpha, i \rangle \models_{\text{CLTL}} \exists_x F & \text{iff there is an } x\text{-variant } \alpha' \text{ of } \alpha \text{ s.t. } \langle \alpha', i \rangle \models_{\text{CLTL}} F
\end{array}$$

Define $\llbracket F \rrbracket = \{\alpha \mid \alpha \models_{\text{CLTL}} F\}$. We say that F is **CLTL** valid iff $\llbracket F \rrbracket = \mathcal{C}^\omega$, and that F is **CLTL** satisfiable iff $\llbracket F \rrbracket \neq \emptyset$.

Next we explain why the operators of the constraint system should not be confused with those of the temporal logic (i.e. the dotted notation). A temporal formula F expresses properties over sequences of constraints. As a state formula, c expresses a property which is satisfied only by those $e.\alpha'$ such that $e \models c$ holds. Therefore, the state formula **false** (and consequently $\square \mathbf{false}$) has at least one sequence that satisfies it (e.g. \mathbf{false}^ω). On the contrary, the temporal formula **false** has

no models whatsoever. Something similar happens with the disjunction and negation operators. In contrast, the formula $c \dot{\wedge} d$ and the atomic proposition $c \wedge d$ have the same models since $e \models (c \wedge d)$ holds if and only if both $e \models c$ and $e \models d$ hold.

5.1.3 Specification Example

Now we shall illustrate how specify a property using the logic presented in this chapter. For this purpose we shall use as reference the `tcc` program in Figure 3.3. Remember that such program has a process which repeatedly checks if the information `in=true` is available. If the information is available in the current time unit, then the process tell that `x=2` in the next time unit or `x=1` otherwise. Therefore, we could check if it true that when the information `in=true` is available, then the information `x=2` will be available in the next time unit. Formula 5.1 represents the above property.

$$\varphi = \Box((in = \mathbf{true}) \dot{\Rightarrow} \circ(x = 2)) \quad (5.1)$$

5.2 Summary

In this chapter we presented a temporal logic which allows us to specify properties of `tcc` systems. In particular, this logic is based on sequences of constraints, and it is used to reason about `ntcc` processes which is an extension of `tcc`. Furthermore, we expressed a property of the program in Figure 3.3 which we shall verify in Chapter 6.

The Model Checking Algorithm

In Chapter 4 we defined a structure called *tcc Structure* which allows us to model the behavior of a system specified in `tcc`. Furthermore, we defined a procedure to construct the model of the system from a `tcc` specification. Then, in Chapter 5 we studied a linear-temporal logic which permits to express temporal properties over constraints and to reason about `tcc` programs. In this chapter, we shall address with the third and last phase of the model checking technique which consists in defining an algorithm that checks if a given temporal formula is satisfied by the model.

We shall start by defining how to construct the *closure* of a formula. Our definition is based on the ideas presented by Manna and Pnueli in [MP95]. This closure is reminiscent to the Fischer-Ladner's one [FL79]. We then define how to construct a graph structure called *model checking graph* which allows to verify if the property is satisfied or not by the system. This graph is built combining the nodes of a `tcc` Structure and the closure of a formula. Our algorithm is based on the classical tableau algorithm for the LTL model checking problem [LP85]. Hence, if we intent to prove that the model satisfies the formula ϕ , then we must construct the model checking graph with the closure of the negated formula (i.e. $\neg\phi$). Finally, we describe the properties that the model checking graph must fulfill to decide if the model satisfies or not the property. Furthermore, we present two examples to show the two possible results of the algorithm.

6.1 The Closure of the Formula

Firstly, given the formula ϕ we have to compute the closure $CL(\phi)$. The closure of a formula φ , $CL(\varphi)$, contains the sub-formulas whose truth values can influence the truth value of φ [CGP99, MP95, FL79]. More precisely, it is the smallest set of formulas satisfying the following conditions:

1. $\varphi \in CL(\varphi)$,
2. $\neg\varphi_1 \in CL(\varphi)$, iff $\varphi_1 \in CL(\varphi)$,
3. if $\varphi_1 \wedge \varphi_2 \in CL(\varphi)$, then $\varphi_1, \varphi_2 \in CL(\varphi)$,
4. if $\varphi_1 \vee \varphi_2 \in CL(\varphi)$, then $\varphi_1, \varphi_2 \in CL(\varphi)$,
5. if $\exists_x \varphi_1 \in CL(\varphi)$, then $\varphi_1 \in CL(\varphi)$,
6. if $\circ\varphi_1 \in CL(\varphi)$, then $\varphi_1 \in CL(\varphi)$,
7. if $\neg \circ \varphi_1 \in CL(\varphi)$, then $\circ\neg\varphi_1 \in CL(\varphi)$,
8. if $\square\varphi_1 \in CL(\varphi)$, then $\varphi_1, \circ\square\varphi_1 \in CL(\varphi)$,

9. if $\diamond\varphi_1 \in CL(\varphi)$, then $\varphi_1, \circ\diamond\varphi_1 \in CL(\varphi)$

Note that in the case of $\dot{\neg}\circ\varphi_1$ it is necessary to introduce the formula $\circ\dot{\neg}\varphi_1$ which cannot be generated by the other rules. To keep the closure finite, we assume that $\dot{\neg}\dot{\neg}\varphi = \varphi$.

6.1.1 Closure Example

We now give an example illustrating the closure of the resulting formula from the negation of Formula 5.1. For convenience, we change the implication operator into a disjunction and we use the duality of the box operator:

$$\begin{aligned}\varphi &= \Box((in = \mathbf{true}) \Rightarrow \circ(x = 2)) \\ &= \Box(\dot{\neg}(in = \mathbf{true}) \dot{\vee} \circ(x = 2))\end{aligned}\tag{6.1}$$

$$\begin{aligned}\dot{\neg}\varphi &= \dot{\neg}\Box(\dot{\neg}(in = \mathbf{true}) \dot{\vee} \circ(x = 2)) \\ &= \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg}\circ(x = 2))\end{aligned}\tag{6.2}$$

Next we show the closure of Formula 6.2.

$$\begin{aligned}CL(\dot{\neg}\varphi) &= \{\diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg}\circ(x = 2)), \\ &\quad \dot{\neg}\diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg}\circ(x = 2)), \\ &\quad \circ\diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg}\circ(x = 2)), \\ &\quad \dot{\neg}\circ\diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg}\circ(x = 2)), \\ &\quad \circ\dot{\neg}\diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg}\circ(x = 2)), \\ &\quad (in = \mathbf{true}) \dot{\wedge} \dot{\neg}\circ(x = 2), \\ &\quad \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg}\circ(x = 2)), \\ &\quad \dot{\neg}\circ(x = 2), \circ\dot{\neg}(x = 2), \\ &\quad \circ(x = 2), (x = 2), \dot{\neg}(x = 2), \\ &\quad (in = \mathbf{true}), \dot{\neg}(in = \mathbf{true}) \\ &\quad \}\end{aligned}$$

6.2 The Model-Checking Graph

In this section we define the structure called *model checking graph* which is essential to check if a model satisfies a formula. This structure is a directed graph derived from a temporal formula and the model of the system.

To determine that a formula ϕ is satisfied by a model, we must construct the graph for the negation of the formula ϕ (i.e. $\neg\phi$). If we prove that there is no computation of the system which satisfies the negated formula, then we are proving that the formula is satisfied for all the computations. This will be discussed in more detail in Section 6.3.

The following definition formalize the construction of the model checking graph.

Definition 6.1 (Model-Checking Graph). Let φ be a formula, $CL(\varphi)$ be the closure of φ as defined in Section 6.1 and Z the **tcc** Structure constructed following the algorithm described in Section 4.3. 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 Z and \mathcal{Q}_n is a subset of $CL(\varphi)$ and the atomic propositions such that the following conditions are satisfied:

1. for every atomic proposition $p, p \in \mathcal{Q}_n$ iff $p \in C(s_n)$,
2. for every $\dot{\exists}_x \varphi_1 \in CL(\varphi), \dot{\exists}_x \varphi_1 \in \mathcal{Q}_n$ iff $\dot{\exists}_x \varphi_1 \in C(s_n)$,
3. for every $\varphi_1 \in CL(\varphi), \varphi_1 \in \mathcal{Q}_n$ iff $\neg \varphi_1 \notin \mathcal{Q}_n$,
4. for every $\varphi_1 \dot{\wedge} \varphi_2 \in CL(\varphi), \varphi_1 \dot{\wedge} \varphi_2 \in \mathcal{Q}_n$ iff $\varphi_1 \in \mathcal{Q}_n$ and $\varphi_2 \in \mathcal{Q}_n$,
5. for every $\varphi_1 \dot{\vee} \varphi_2 \in CL(\varphi), \varphi_1 \dot{\vee} \varphi_2 \in \mathcal{Q}_n$ iff $\varphi_1 \in \mathcal{Q}_n$ or $\varphi_2 \in \mathcal{Q}_n$,
6. for every $\dot{\circ} \varphi_1 \in CL(\varphi), \dot{\circ} \varphi_1 \in \mathcal{Q}_n$ iff $\circ \neg \varphi_1 \in \mathcal{Q}_n$,
7. for every $\square \varphi_1 \in CL(\varphi), \square \varphi_1 \in \mathcal{Q}_n$ iff $\varphi_1 \in \mathcal{Q}_n$ and $\circ \square \varphi_1 \in \mathcal{Q}_n$,
8. for every $\diamond \varphi_1 \in CL(\varphi), \diamond \varphi_1 \in \mathcal{Q}_n$ iff $\varphi_1 \in \mathcal{Q}_n$ or $\circ \diamond \varphi_1 \in \mathcal{Q}_n$.

An edge in the graph is defined as follows: there will be an edge from one node $n_1 = (s_1, \mathcal{Q}_1)$ to another node $n_2 = (s_2, \mathcal{Q}_2)$ iff there is an arc from the node s_1 to the node s_2 in Z and for every formula $\circ \varphi_1 \in CL(\varphi), \circ \varphi_1 \in \mathcal{Q}_1$ iff $\varphi_1 \in \mathcal{Q}_2$. This means that the next state s_2 must satisfy ϕ if s_1 satisfies $\circ \phi$.

Intuitively, for each node of the model-checking graph, in \mathcal{Q} we have the largest consistent set of formulas that is also consistent with the *store* (function C) of the states of the **tcc** Structure. Moreover, two nodes of the graph are related if the temporal formulas in their \mathcal{Q} sets are consistent.

For each node s_i of the **tcc** Structure many nodes are generated in the graph with a different consistent set of formulas derived from $C(s_i)$ and the closure of the formula $CL(\phi)$. We exemplify this in the following section.

6.2.1 Model Checking Graph Example

We next illustrate the construction of the model checking graph. In the following example, we construct the graph which we shall use in the next section in order to determine if Formula 6.1 is satisfied by the model in Figure 4.7. Recall that such graph was generated from the specification in Figure 3.3.

Firstly, we take the **tcc** Structure shown in Figure 4.7 and the closure set of Formula 6.2 shown in the example of the previous section. Remind that if we want to prove that the formula ϕ is satisfied by the model, then we must generate the closure of the negated formula (i.e. $\neg \phi$). After that, using the Definition 6.1 we generate all the possible nodes from the nodes of the **tcc** Structure and the formulas of the closure.

Next we show all the nodes generated.

$n_1 = (s_1, Q_1)$ where

$$Q_1 = \{in = \mathbf{true}, x = 2, \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)) \\ \}$$

$n_3 = (s_1, Q_3)$ where

$$Q_3 = \{in = \mathbf{true}, x = 2, \\ \circ \dot{\neg}(x = 2), \dot{\neg} \circ(x = 2), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \}$$

$n_5 = (s_1, Q_5)$ where

$$Q_5 = \{in = \mathbf{true}, \dot{\neg}(x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)) \\ \}$$

$n_2 = (s_1, Q_2)$ where

$$Q_2 = \{in = \mathbf{true}, x = 2, \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)) \\ \}$$

$n_4 = (s_1, Q_4)$ where

$$Q_4 = \{in = \mathbf{true}, x = 2, \\ \circ \dot{\neg}(x = 2), \dot{\neg} \circ(x = 2), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)) \\ \}$$

$n_6 = (s_1, Q_6)$ where

$$Q_6 = \{in = \mathbf{true}, \dot{\neg}(x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)) \\ \}$$

$n_7 = (s_1, Q_7)$ where

$$Q_7 = \{in = \mathbf{true}, \dot{\circ}(x = 2), \\ \circ \dot{\circ}(x = 2), \dot{\circ} \circ (x = 2), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \}$$

$n_9 = (s_2, Q_9)$ where

$$Q_9 = \{in = \mathbf{true}, (x = 2), \circ(x = 2), \\ \dot{\circ}((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \}$$

$n_{11} = (s_2, Q_{11})$ where

$$Q_{11} = \{in = \mathbf{true}, (x = 2), \\ \dot{\circ} \circ (x = 2), \circ \dot{\circ}(x = 2), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \}$$

$n_8 = (s_1, Q_8)$ where

$$Q_8 = \{in = \mathbf{true}, \dot{\circ}(x = 2), \\ \circ \dot{\circ}(x = 2), \dot{\circ} \circ (x = 2), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2), \\ \circ \dot{\circ} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \dot{\circ} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)) \\ \}$$

$n_{10} = (s_2, Q_{10})$ where

$$Q_{10} = \{in = \mathbf{true}, (x = 2), \circ(x = 2), \\ \dot{\circ}((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \circ \dot{\circ} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \dot{\circ} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \dot{\circ} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)) \\ \}$$

$n_{12} = (s_2, Q_{12})$ where

$$Q_{12} = \{in = \mathbf{true}, (x = 2), \\ \dot{\circ} \circ (x = 2), \circ \dot{\circ}(x = 2), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2), \\ \circ \dot{\circ} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \dot{\circ} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\circ} \circ (x = 2)) \\ \}$$

$n_{13} = (s_3, Q_{13})$ where

$$Q_{13} = \{ \dot{\neg}(in = \mathbf{true}), (x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \}$$

$n_{15} = (s_3, Q_{15})$ where

$$Q_{15} = \{ \dot{\neg}(in = \mathbf{true}), (x = 2), \\ \dot{\neg} \circ(x = 2), \circ \dot{\neg}(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \}$$

$n_{17} = (s_4, Q_{17})$ where

$$Q_{17} = \{ \dot{\neg}(in = \mathbf{true}), (x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \}$$

$n_{14} = (s_3, Q_{14})$ where

$$Q_{14} = \{ \dot{\neg}(in = \mathbf{true}), (x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)) \\ \}$$

$n_{16} = (s_3, Q_{16})$ where

$$Q_{16} = \{ \dot{\neg}(in = \mathbf{true}), (x = 2), \\ \dot{\neg} \circ(x = 2), \circ \dot{\neg}(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)) \\ \}$$

$n_{18} = (s_4, Q_{18})$ where

$$Q_{18} = \{ \dot{\neg}(in = \mathbf{true}), (x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 2)), \\ \}$$

$n_{19} = (s_4, Q_{19})$ where

$$Q_{19} = \{ \dot{\neg}(in = \mathbf{true}), (x = 2), \\ \circ \dot{\neg}(x = 2), \dot{\neg} \circ (x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \}$$

$n_{21} = (s_4, Q_{21})$ where

$$Q_{21} = \{ \dot{\neg}(in = \mathbf{true}), \dot{\neg}(x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \}$$

$n_{23} = (s_4, Q_{23})$ where

$$Q_{23} = \{ \dot{\neg}(in = \mathbf{true}), \dot{\neg}(x = 2), \\ \circ \dot{\neg}(x = 2), \dot{\neg} \circ (x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \}$$

$n_{20} = (s_4, Q_{20})$ where

$$Q_{20} = \{ \dot{\neg}(in = \mathbf{true}), (x = 2), \\ \circ \dot{\neg}(x = 2), \dot{\neg} \circ (x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \}$$

$n_{22} = (s_4, Q_{22})$ where

$$Q_{22} = \{ \dot{\neg}(in = \mathbf{true}), \dot{\neg}(x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \}$$

$n_{24} = (s_4, Q_{24})$ where

$$Q_{24} = \{ \dot{\neg}(in = \mathbf{true}), \dot{\neg}(x = 2), \\ \circ \dot{\neg}(x = 2), \dot{\neg} \circ (x = 2), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 2)), \\ \}$$

$n_{25} = (s_5, Q_{25})$ where

$$Q_{25} = \{ (in = \mathbf{true}), (x = 1), \\ \dot{\neg}(x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \circ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \}$$

$n_{27} = (s_5, Q_{27})$ where

$$Q_{27} = \{ (in = \mathbf{true}), (x = 1), \dot{\neg}(x = 2), \\ \dot{\neg} \circ(x = 2), \circ \dot{\neg}(x = 2), \\ (in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2), \\ \circ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \}$$

$n_{29} = (s_6, Q_{29})$ where

$$Q_{29} = \{ \dot{\neg}(in = \mathbf{true}), (x = 1), \\ \dot{\neg}(x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \circ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \}$$

$n_{26} = (s_5, Q_{26})$ where

$$Q_{26} = \{ (in = \mathbf{true}), (x = 1), \\ \dot{\neg}(x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)) \\ \}$$

$n_{28} = (s_5, Q_{28})$ where

$$Q_{28} = \{ (in = \mathbf{true}), (x = 1), \dot{\neg}(x = 2), \\ \dot{\neg} \circ(x = 2), \circ \dot{\neg}(x = 2), \\ (in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)) \\ \}$$

$n_{30} = (s_6, Q_{30})$ where

$$Q_{30} = \{ \dot{\neg}(in = \mathbf{true}), (x = 1), \\ \dot{\neg}(x = 2), \circ(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2)) \\ \}$$

$n_{31} = (s_6, Q_{31})$ where

$$Q_{31} = \{ \dot{\neg}(in = \mathbf{true}), (x = 1), \dot{\neg}(x = 2), \\ \dot{\neg} \circ (x = 2), \circ \dot{\neg}(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \wedge \dot{\neg} \circ (x = 2)), \\ \circ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ (x = 2)), \\ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ (x = 2)), \\ \}$$

$n_{32} = (s_6, Q_{32})$ where

$$Q_{32} = \{ \dot{\neg}(in = \mathbf{true}), (x = 1), \dot{\neg}(x = 2), \\ \dot{\neg} \circ (x = 2), \circ \dot{\neg}(x = 2), \\ \dot{\neg}((in = \mathbf{true}) \wedge \dot{\neg} \circ (x = 2)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ (x = 2)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ (x = 2)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ (x = 2)) \\ \}$$

For instance, nodes n_{31} and n_{32} are generated from the state s_6 of the model, but n_{31} satisfies $\diamond((in = \mathbf{true}) \wedge \neg \circ (x = 2))$ while n_{32} does not.

Then, we define the arcs of the graph following the definition of the model-checking graph. In Figure 6.1, we show the resulting model checking graph for the `tcc` Structure shown in Figure 4.7 and Formula 6.2.

6.3 The Searching Algorithm

In this section we define the algorithm that allows to determine if a model satisfies or not a formula. Our algorithm is based on the classical approach [CGP99, MP95, LP85]: to prove that a property is satisfied, it suffices to prove that there is no path in the model checking graph satisfying the negation of the formula. Before defining our algorithm, let us introduce some necessary definitions.

Firstly, we need to define what is a strongly connected component in a graph.

Definition 6.2 (Strongly Connected Component [MP95]). Given a graph G , we define a Strongly Connected Component (SCC) S as a maximal subgraph of G such that for every two distinct nodes $A, B \in S$, there exists a path from A to B that passes through nodes of S .

We say that S is *transient* if it consists of a single node that is not connected to itself.

Now we define a kind of SCC called *Self-fulfilling SCC* which satisfies some properties. The following definition is based on the idea of *promising formulas* presented in [MP95]. A formula of the form $\diamond\phi$ can be viewed as a promise that ϕ will eventually hold.

Definition 6.3 (Self-fulfilling SCC). Given a model-checking graph G , a self-fulfilling strongly connected component C is defined as a non-transient strongly connected component in G that satisfies that for every node n in C and for every $\diamond\phi \in \mathcal{Q}_n$ there exists a node m in C such that $\phi \in \mathcal{Q}_m$.

Notice that assuming that ϕ is in the same SCC is important. It could be the case that $\diamond\phi$ holds but ϕ is satisfied "outside" the SCC.

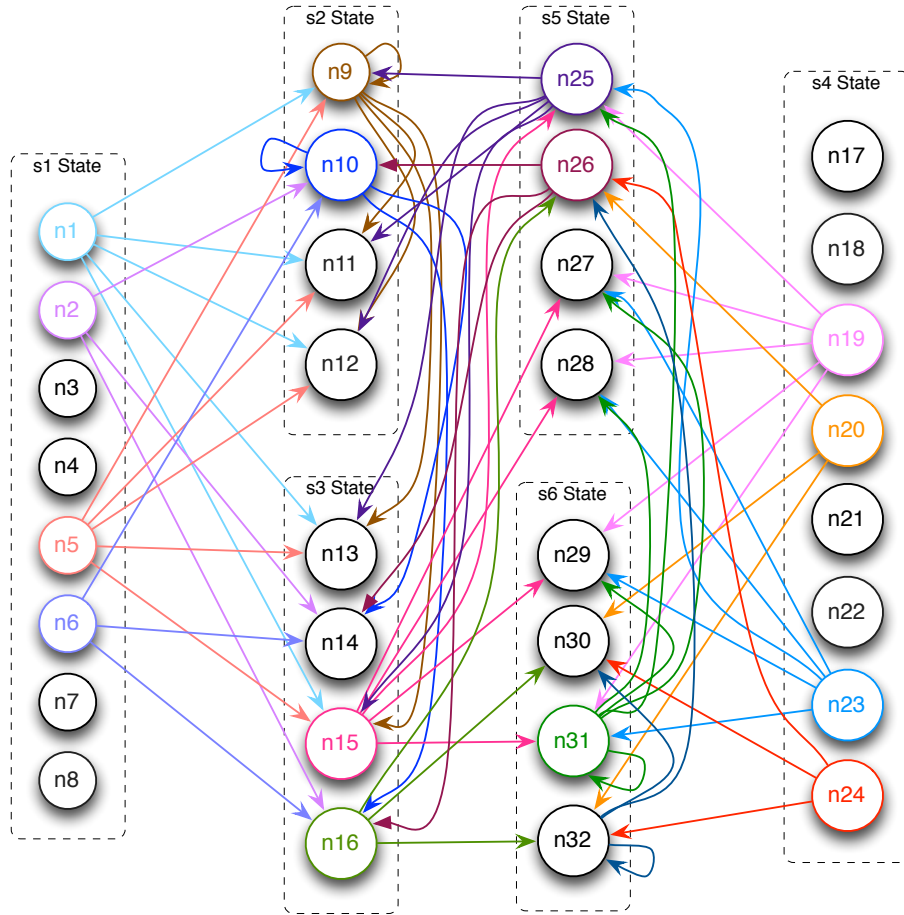


Figure 6.1: Model checking graph for the tcc Structure shown in Figure 4.7 and the formula 6.2

We say that a sequence is an *eventually sequence* if it is an infinite path in a model checking graph G such that if there exists a node n in the path with $\diamond\phi \in \mathcal{Q}_n$, then there exists another node m in the same path reachable from n along the path, such that $\phi \in \mathcal{Q}_m$. Now we can define our model checking algorithm.

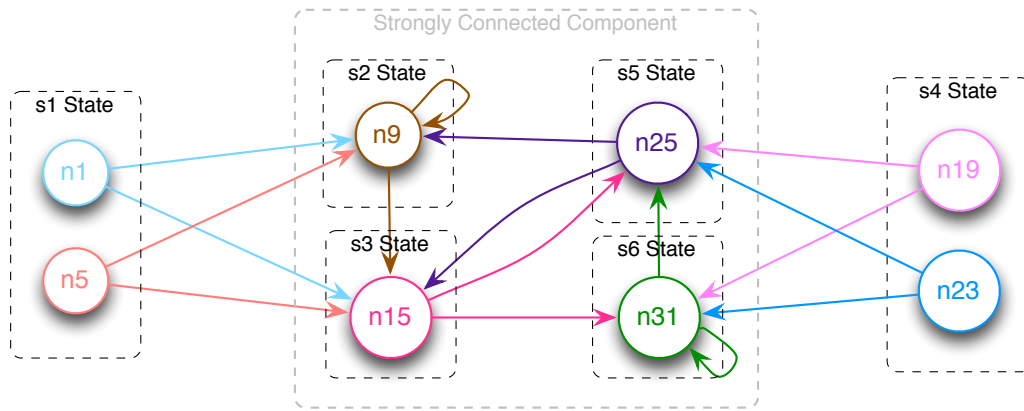
1. Construct the model checking graph using the negation of ϕ ($-\phi$) and the model of the system.
2. Look for a sequence such that starting from an initial node of the graph that satisfies the negation of ϕ , it reaches a *self-fulfilling strongly connected component*.
3. If we find a self-fulfilling SCC in the model-checking graph, then the system satisfies the property represented by the negated formula. Thus, we need to prove that such self-fulfilling SCC does not exist in order to prove that the original formula is satisfied by the model.

Having defined the model checking algorithm, we implemented a prototype of the algorithm.

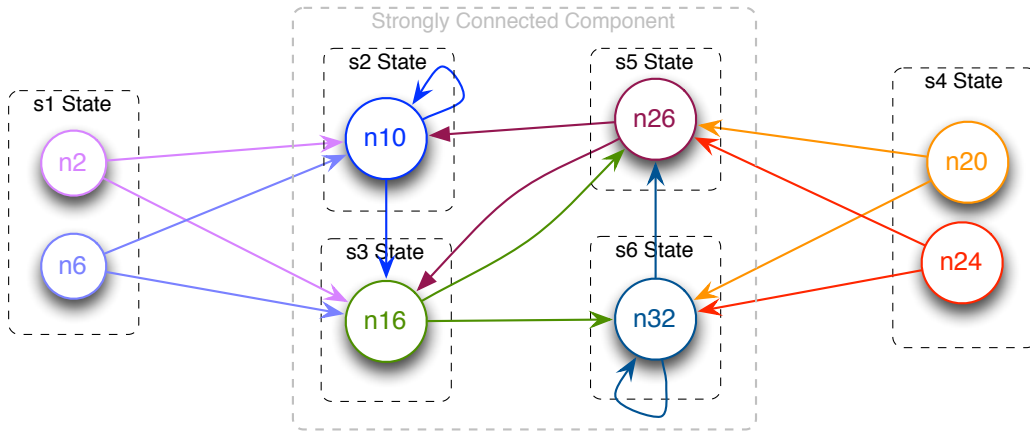
This prototype is described in Chapter 7. We apply our implementation of the model checking algorithm on the examples of Subsection 6.3.1 and Section 6.4.

6.3.1 Searching Algorithm Example

The following example illustrates the model checking algorithm. We use the model checking graph shown in Figure 6.1 in order to prove that Formula 6.1 is satisfied by the model shown in Figure 4.7. Next, in Figure 6.2(a) and 6.2(b) we show the non-transient strongly connected components of the model checking graph.



(a) First non-transient SCC



(b) Second non-transient SCC

Figure 6.2: Non-transient SCCs of the model checking graph shown in Figure 6.1

Notice that the SCC in Figure 6.2(a) is not self-fulfilling because the node n_9 satisfies the formula $\circ\Diamond((in = \text{true}) \wedge \neg \circ(x = 2))$ and any node in the SCC eventually holds the formula promised (i.e.

$\diamond((in = \mathbf{true}) \wedge \neg \circ(x = 2))$). Moreover, the SCC in Figure 6.2(b) has not an initial node satisfying the negated formula (i.e. $\diamond((in = \mathbf{true}) \wedge \dot{\neg} \circ(x = 2))$). Thus, the model does not satisfy the negated formula (i.e. the model satisfies the original formula $\Box(\dot{\neg}(in = \mathbf{true}) \dot{\vee} \circ(x = 2))$).

6.4 Model Checking Example

In this section we illustrate other application of our model checking algorithm. We shall prove that Formula 6.3 is not satisfied by the model shown in Figure 4.7. As in the previous example, here we change the implication operator into a disjunction and we use the duality of the box operator.

$$\begin{aligned} \varphi &= \Box((in = \mathbf{true}) \Rightarrow \circ(x = 1)) \\ &= \Box(\dot{\neg}(in = \mathbf{true}) \dot{\vee} \circ(x = 1)) \end{aligned} \quad (6.3)$$

We start by calculating the closure of the resulting formula from the negation of Formula 6.3.

$$\begin{aligned} \dot{\neg}\varphi &= \dot{\neg}\Box(\dot{\neg}(in = \mathbf{true}) \dot{\vee} \circ(x = 1)) \\ &= \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)) \end{aligned} \quad (6.4)$$

$$\begin{aligned} CL(\dot{\neg}\varphi) &= \{ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ &\quad \dot{\neg}\diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ &\quad \circ\diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ &\quad \dot{\neg}\circ\diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ &\quad \circ\dot{\neg}\diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ &\quad (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1), \\ &\quad \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ &\quad \dot{\neg}\circ(x = 1), \circ\dot{\neg}(x = 1), \\ &\quad \circ(x = 1), (x = 1), \dot{\neg}(x = 1), \\ &\quad (in = \mathbf{true}), \dot{\neg}(in = \mathbf{true}) \\ &\quad \} \end{aligned}$$

Then, we generate all the possible nodes of the model checking graph using Definition 6.1. We next show the nodes generated from the model of the system and the closure of Formula 6.4.

$n_1 = (s_1, Q_1)$ where

$$Q_1 = \{in = \mathbf{true}, x = 1, \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)) \\ \}$$

$n_3 = (s_1, Q_3)$ where

$$Q_3 = \{in = \mathbf{true}, x = 1, \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ(x = 1), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_5 = (s_1, Q_5)$ where

$$Q_5 = \{in = \mathbf{true}, \dot{\neg}(x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)) \\ \}$$

$n_2 = (s_1, Q_2)$ where

$$Q_2 = \{in = \mathbf{true}, x = 1, \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)) \\ \}$$

$n_4 = (s_1, Q_4)$ where

$$Q_4 = \{in = \mathbf{true}, x = 1, \\ \circ \dot{\neg}(x = 1) \dot{\neg} \circ(x = 1), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)) \\ \}$$

$n_6 = (s_1, Q_6)$ where

$$Q_6 = \{in = \mathbf{true}, \dot{\neg}(x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)) \\ \}$$

$n_7 = (s_1, Q_7)$ where

$$Q_7 = \{ \begin{array}{l} in = \mathbf{true}, \dot{\neg}(x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \} \end{array}$$

$n_9 = (s_2, Q_9)$ where

$$Q_9 = \{ \begin{array}{l} in = \mathbf{true}, x = 2, \\ \dot{\neg}(x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \} \end{array}$$

$n_{11} = (s_2, Q_{11})$ where

$$Q_{11} = \{ \begin{array}{l} in = \mathbf{true}, x = 2, \dot{\neg}(x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \} \end{array}$$

$n_8 = (s_1, Q_8)$ where

$$Q_8 = \{ \begin{array}{l} in = \mathbf{true}, \dot{\neg}(x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)) \\ \} \end{array}$$

$n_{10} = (s_2, Q_{10})$ where

$$Q_{10} = \{ \begin{array}{l} in = \mathbf{true}, x = 2, \\ \dot{\neg}(x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)) \\ \} \end{array}$$

$n_{12} = (s_2, Q_{12})$ where

$$Q_{12} = \{ \begin{array}{l} in = \mathbf{true}, x = 2, \dot{\neg}(x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)) \\ \} \end{array}$$

$n_{13} = (s_3, Q_{13})$ where

$$Q_{13} = \{ \dot{\neg}(in = \mathbf{true}), x = 2, \\ \dot{\neg}(x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{15} = (s_3, Q_{15})$ where

$$Q_{15} = \{ \dot{\neg}(in = \mathbf{true}), x = 2, \dot{\neg}(x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{17} = (s_4, Q_{17})$ where

$$Q_{17} = \{ \dot{\neg}(in = \mathbf{true}), (x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{14} = (s_3, Q_{14})$ where

$$Q_{14} = \{ \dot{\neg}(in = \mathbf{true}), x = 2, \\ \dot{\neg}(x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)) \\ \}$$

$n_{16} = (s_3, Q_{16})$ where

$$Q_{16} = \{ \dot{\neg}(in = \mathbf{true}), x = 2, \dot{\neg}(x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)) \\ \}$$

$n_{18} = (s_4, Q_{18})$ where

$$Q_{18} = \{ \dot{\neg}(in = \mathbf{true}), (x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{19} = (s_4, Q_{19})$ where

$$Q_{19} = \{ \dot{\neg}(in = \mathbf{true}), (x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \}$$

$n_{21} = (s_4, Q_{21})$ where

$$Q_{21} = \{ \dot{\neg}(in = \mathbf{true}), \dot{\neg}(x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \}$$

$n_{23} = (s_4, Q_{23})$ where

$$Q_{23} = \{ \dot{\neg}(in = \mathbf{true}), \dot{\neg}(x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \}$$

$n_{20} = (s_4, Q_{20})$ where

$$Q_{20} = \{ \dot{\neg}(in = \mathbf{true}), (x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \}$$

$n_{22} = (s_4, Q_{22})$ where

$$Q_{22} = \{ \dot{\neg}(in = \mathbf{true}), \dot{\neg}(x = 1), \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \}$$

$n_{24} = (s_4, Q_{24})$ where

$$Q_{24} = \{ \dot{\neg}(in = \mathbf{true}), \dot{\neg}(x = 1), \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \}$$

$n_{25} = (s_5, Q_{25})$ where

$$Q_{25} = \{ in = \mathbf{true}, x = 1, \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{27} = (s_5, Q_{27})$ where

$$Q_{27} = \{ in = \mathbf{true}, x = 1, \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ(x = 1), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{29} = (s_6, Q_{29})$ where

$$Q_{29} = \{ \dot{\neg}(in = \mathbf{true}), x = 1, \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{26} = (s_5, Q_{26})$ where

$$Q_{26} = \{ in = \mathbf{true}, x = 1, \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{28} = (s_5, Q_{28})$ where

$$Q_{28} = \{ in = \mathbf{true}, x = 1, \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ(x = 1), \\ (in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{30} = (s_6, Q_{30})$ where

$$Q_{30} = \{ \dot{\neg}(in = \mathbf{true}), x = 1, \circ(x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \circ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \circ \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \dot{\neg} \diamond((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ(x = 1)), \\ \}$$

$n_{31} = (s_6, Q_{31})$ where

$$Q_{31} = \{ \dot{\neg}(in = \mathbf{true}), x = 1, \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \}$$

$n_{32} = (s_6, Q_{32})$ where

$$Q_{32} = \{ \dot{\neg}(in = \mathbf{true}), x = 1, \\ \circ \dot{\neg}(x = 1), \dot{\neg} \circ (x = 1), \\ \dot{\neg}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \circ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \circ \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \dot{\neg} \dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1)), \\ \}$$

Using the definition of the model checking graph, we define the arcs of the graph. In Figure 6.3, we show the resulting model checking graph for the `tcc` Structure shown in Figure 4.7 and Formula 6.4.

Now we must look for a self-fulfilling SCC in the model checking graph generated above. In Figure 6.4, we show the only non-transient SCC of the graph. Notice that it is a self-fulfilling SCC because all the promised formulas in the subgraph are eventually satisfied. Moreover, it has an initial node which satisfies the negated formula (i.e. $\dot{\diamond}((in = \mathbf{true}) \dot{\wedge} \dot{\neg} \circ (x = 1))$). Thus, the model satisfies the negated formula (i.e. the model does not satisfy the original property $\Box(\dot{\neg}(in = \mathbf{true}) \dot{\vee} \circ (x = 1))$).

6.5 Summary

In this chapter, we defined the algorithm which allows to determine if a formula is satisfied or not by a model. This algorithm uses a structure called *model checking graph* which is constructed from a formula and the model of the system. We defined how to construct this graph and how to calculate the closure of a formula. The key idea of the algorithm is to construct the model checking graph using the negation of the formula and the model of the system, and then to look for a sequence such that starting from an initial node in the graph, it reaches a self-fulfilling strongly connected component. If we do not find the path, then we prove that the model does not satisfy the negated formula. This is equivalent to prove that the model satisfies the original formula.

Furthermore, we presented two examples that exhibit the two possible outputs of the algorithm. Each example shows in detail the different steps of the algorithm.

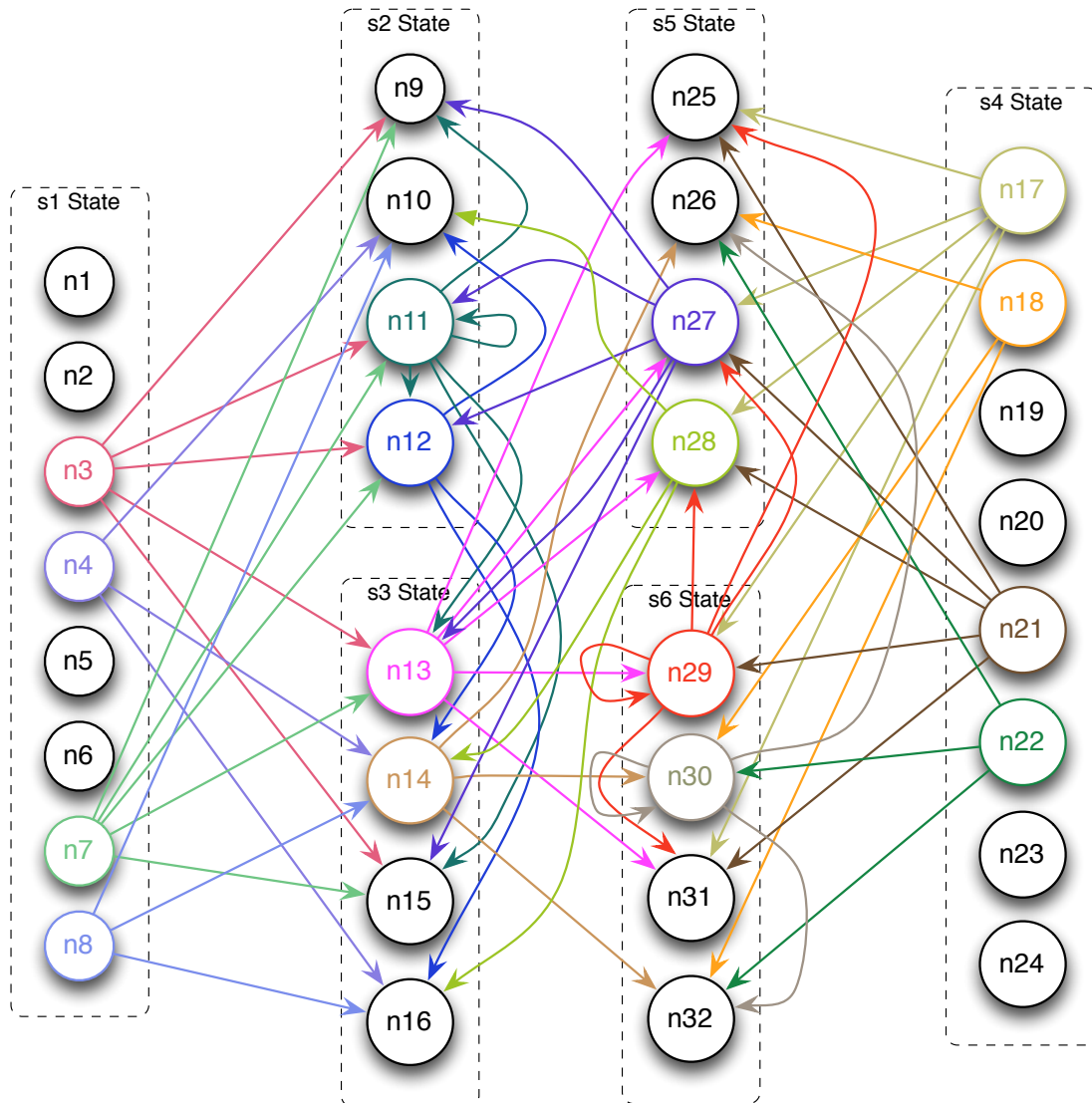


Figure 6.3: Model checking graph for the tcc Structure shown in Figure 4.7 and Formula 6.4

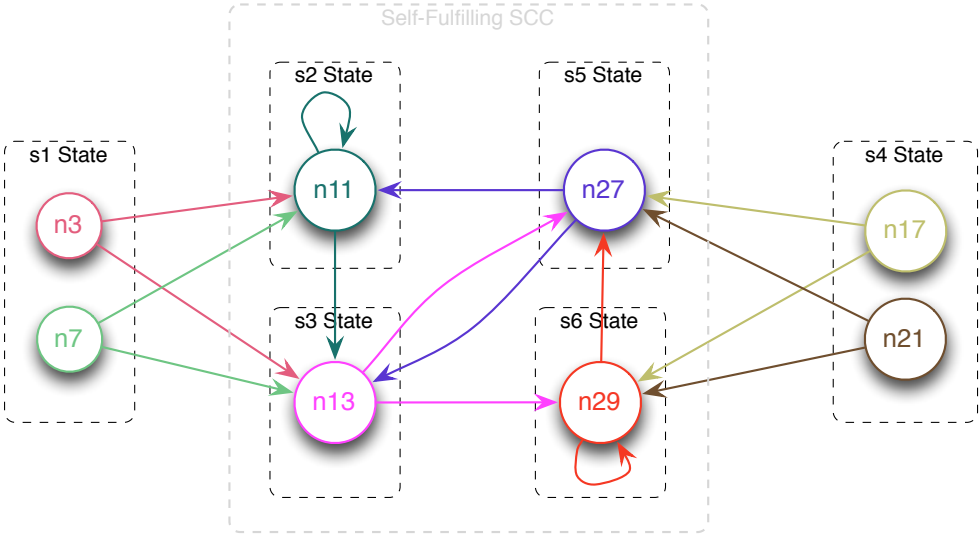


Figure 6.4: Non-transient SCC of the model checking graph shown in Figure 6.3

A Prototypical Tool

In the previous chapter we described the model checking algorithm which determines whether the model of the system constructed from a `tcc` specification satisfies a temporal formula. Now we focus on developing a tool to run the algorithm automatically. In this chapter, we shall describe a prototype tool that implements our model checking algorithm.

The algorithm takes as input the model of the system and the formula to be verified. Thus, we start by specifying the structures to define them. We then show the output of the algorithm and we describe the functions that calculate the closure of a formula, the model checking nodes, the model checking graph, and the strongly connected components. We use the Python programming language to implement the algorithm.

7.1 Inputs

The model checking algorithm receives as input the model of the system and the temporal formula to be verified. We then need a structure to represent a `tcc` Structure and a temporal formula using the proposed logic. Recall that the `tcc` Structure is formally defined in Chapter 4 and the syntax of the logic in Chapter 5.

7.1.1 Property

First of all, we define a structure to represent temporal formulas and constraints. We use the syntax presented in Definition 5.1.

Let us first introduce the encoding of the logical operators we use in our implementation.

- Always: \square
- Sometimes: $\langle \rangle$
- Next: \circ
- Negation: \sim
- Or: \vee
- And: \wedge

We represent a temporal formula as a binary abstract syntax tree (AST)¹. For example, the formula $\diamond((\text{in} = \text{true}) \wedge \neg \circ (\text{x} = 1))$, would be represented as such:

¹http://en.wikipedia.org/wiki/Abstract_syntax_tree

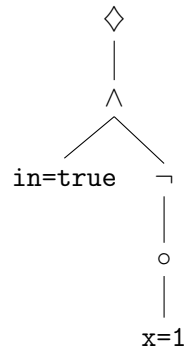


Figure 7.1: Representation of $\diamond((\text{in} = \text{true}) \wedge \neg o(x = 1))$ using AST

Let us illustrate the above notion in Python.

```
>>> phi = Formula({"<>": {"^": {"": "in=true", "~": {"o": "x=1"}}}})
```

Figure 7.2: Example of a formula in Python

The `Formula` class implements some operations over a formula such as getting the main connector.

7.1.2 System Model

In the following we define a structure to represent a `tcc` node. Recall that a node has constraints (*store*), internal labels, temporal labels. Moreover, it is related to other nodes and it can be an initial node. In Figure 7.3, we show a schematic representation of the structure that describes a `tcc` node.

tcc node	
store:	<i>(list of Formulas)</i>
internal:	<i>(list of strings)</i>
temporal:	<i>(list of strings)</i>
edges:	<i>(list of integers)</i>
initial:	<i>(boolean)</i>

Figure 7.3: Representation of a `tcc` node

In Figure 7.4, we show the encoding of the node `s1` of the `tcc` Structure in Figure 4.7.

Now we can represent a system model. Basically, we assume that a `tcc` Structure is a list of `tcc` nodes. For convenience, we use a dictionary structure which has the number of the node as key and a `tcc` node structure as value. Figure 7.5 shows the encoding of the `tcc` Structure shown in Figure 4.7.

```
>>> {"store": [Formula({"": "in=true"})], "internal": [], "temporal": ["tell4", "p9"], "edges": [2,3], "initial": True}
```

Figure 7.4: Example of a tcc node in Python

```
>>> tcc_structure = {
... 1: {"store": [Formula({"": "in=true"})], "internal": [], "temporal": ["tell4", "p9"], "edges": [2,3], "initial": True},
... 2: {"store": [Formula({"": "x=2"}), Formula({"": "in=true"})], "internal": [], "temporal": ["tell4", "p9"], "edges": [2,3], "initial": False},
... 3: {"store": [Formula({"": "x=2"}), Formula({"~": "in=true"})], "internal": ["nowp2"], "temporal": ["tell7", "p9"], "edges": [5,6], "initial": False},
... 4: {"store": [Formula({"~": "in=true"})], "internal": ["nowp2"], "temporal": ["tell7", "p9"], "edges": [5,6], "initial": True},
... 5: {"store": [Formula({"": "x=1"}), Formula({"": "in=true"})], "internal": [], "temporal": ["tell4", "p9"], "edges": [2,3], "initial": False},
... 6: {"store": [Formula({"": "x=1"}), Formula({"~": "in=true"})], "internal": ["nowp2"], "temporal": ["tell7", "p9"], "edges": [5,6], "initial": False}
... }
```

Figure 7.5: Example of a system model in Python

7.2 Model Checking Function

In this section we show the output of the main function (`modelSatisfiesProperty`), and we describe the auxiliary functions. The main function takes as input a system model and the resulting formula from the negation of the formula to be verified.

In the following example we determine if the Formula 6.3 is satisfied by the model shown in Figure 4.7. Recall that we must enter the negated formula as input to the function. We assume that the variables `phi` and `tcc_structure` are the same as defined in the examples in the previous section.

```
>>> result = modelSatisfiesProperty(phi, tcc_structure)
is Self Fulfilling: True
Initial Nodes Entail Formula: True
>>> print "Model Satisfies Original Formula: ", not result
Model Satisfies Formula: False
```

Figure 7.6: Outcome of the main function

Notice that we negate the outcome of the function. This is because we are verifying the negation of the formula that we want to prove. Furthermore, the model does not satisfy the original formula.

Let us now describe the functions that calculate the main components of the model checking algorithm. We shall introduce the functions following the execution flow of the algorithm.

7.2.1 Closure

First of all, the algorithm must calculate the closure of the formula. We implement the function that calculates the closure of a formula (`getClosure`) following the conditions presented in Section 6.1 in a recursive way. The following example shows the result of calculating the closure of the formula defined in Figure 7.2.

```
>>> closure = []
>>> getClosure(phi,closure)
>>> for formula in closure:
...     print formula.getFormula()
...
{'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}
{'~': {'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}}
{'o': {'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}}
{'~': {'o': {'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}}}}
{'o': {'~': {'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}}}}
{'~': {'': 'in=true', '~': {'o': 'x=1'}}}}
{'~': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}}
{'': 'in=true'}
{'~': 'in=true'}
{'o': 'x=1'}
{'~': {'o': 'x=1'}}
{'o': {'~': 'x=1'}}
{'': 'x=1'}
{'~': 'x=1'}
```

Figure 7.7: Output of `getClosure` function

7.2.2 Model Checking Nodes

Then, the algorithm must generate all the possible nodes of the model checking graph from the `tcc` Structure and the closure of the formula. We implement this function (`getModelCheckingAtoms`) based on Definition 6.1. We next show the nodes generated from the `tcc` Structure defined in Figure 7.5 and the closure calculated in Figure 7.7. To save space we show only two atoms of 32 possible atoms.

```

>>> atoms = getAllAtoms(closure)
>>> model_checking_atoms = getModelCheckingAtoms(tcc_structure, atoms)
>>> for tcc_node in model_checking_atoms.keys():
...     print "tcc State", tcc_node
...     tcc_atoms = model_checking_atoms.get(tcc_node)
...     for atom_index in tcc_atoms.keys():
...         print "Atom ", atom_index
...         for formula in tcc_atoms.get(atom_index):
...             print formula.getFormula()
...         print "\n"
tcc State 1
Atom 1
{'o': {'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}
{'': 'in=true'}
{'o': 'x=1'}
{'': 'x=1'}
{'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}
{'~': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}

Atom 2
{'~': {'o': {'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}}}
{'': 'in=true'}
{'o': 'x=1'}
{'': 'x=1'}
{'o': {'~': {'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}}}
{'~': {'<>': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}}}
{'~': {'~': {'': 'in=true', '~': {'o': 'x=1'}}}}}

```

Figure 7.8: Outcome of `getModelCheckingAtoms` function

7.2.3 Model Checking Graph

Having generated the nodes of the model checking graph, we are ready to construct the graph defining arcs between the nodes previously generated. We implement this function (`getModelCheckingGraph`) following Definition 6.1. The graph is represented by a dictionary structure which has a node as key and the list of its successors as value. In Figure 7.9, we present the model checking graph constructed from the nodes generated in Figure 7.8 and the `tcc` Structure defined in Figure 7.5.

7.2.4 Self-Fulfilling Strongly Connected Components

Finally, the algorithm must look for a path in the model checking graph that, starting from an initial node that satisfies the formula, it reaches a self-fulfilling strongly connected component (SCC). To implement this, we first obtain the SCCs of the graph using the Tarjan's algo-

```

>>> model_checking_graph = getModelCheckingGraph(tcc_structure,
    model_checking_atoms)
>>> model_checking_graph
{1: [], 2: [], 3: [9, 11, 12, 13, 15], 4: [10, 16, 14], 5: [], 6: [], 7:
  [9, 11, 12, 13, 15], 8: [10, 16, 14], 9: [], 10: [], 11: [9, 11, 12,
  13, 15], 12: [10, 16, 14], 13: [25, 27, 28, 29, 31], 14: [26, 32, 30],
  15: [], 16: [], 17: [25, 27, 28, 29, 31], 18: [26, 32, 30], 19: [], 20:
  [], 21: [25, 27, 28, 29, 31], 22: [26, 32, 30], 23: [], 24: [], 25:
  [], 26: [], 27: [9, 11, 12, 13, 15], 28: [10, 16, 14], 29: [25, 27, 28,
  29, 31], 30: [26, 32, 30], 31: [], 32: []}

```

Figure 7.9: Outcome of `getModelCheckingGraph` function

rithm², and we discard the transient SCCs (see Definition 6.2). This is done by the `tarjan` and `getModelCheckingSCCSubgraphs` functions. Then, we check if the SCCs obtained are self-fulfilling SCC (see Definition 6.3) and they have an initial node that satisfies the formula. We implement the functions `isSelfFulfilling` and `initialNodesEntailFormula` to perform these tasks. In Figures 7.10, 7.11 and 7.12, we present the output of the functions listed above.

```

>>> strongly_connected_components = tarjan(model_checking_graph)
>>> sccGraphs = getModelCheckingSCCSubgraphs(strongly_connected_components
    , tcc_structure , model_checking_atoms , model_checking_graph)
>>> sccGraphs[0]
[{3: [11, 13], 7: [11, 13], 11: [11, 13], 13: [27, 29], 17: [27, 29], 21:
  [27, 29], 27: [11, 13], 29: [27, 29]}]

```

Figure 7.10: Output of `getModelCheckingSCCSubgraphs` function

```

>>> initialNodes = getInitialNodes(tcc_structure, model_checking_atoms)
>>> initialNodes
[1, 2, 3, 4, 5, 6, 7, 8, 17, 18, 19, 20, 21, 22, 23, 24]
>>> isSelfFulfilling(sccGraphs[0], initialNodes, model_checking_atoms)
True

```

Figure 7.11: Output of `isSelfFulfilling` function

Notice that the algorithm obtains a self-fulfilling SCC (see Figure 7.11) and it has an initial node that satisfies the formula (see Figure 7.12). For this reason, the output of the main function shown in Figure 7.6 is `True`, thus the model satisfies the formula.

²http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm

```
>>> initialNodesEntailFormula(sccGraphs[0], initialNodes,
    model_checking_atoms, formula)
True
```

Figure 7.12: Output of `initialNodesEntailFormula` function

7.3 Summary

In this chapter we introduced a prototype tool which implements the model checking algorithm presented in Chapter 6. Recall that this algorithm determines if a formula is satisfied by a model. Since the algorithm receives as input the model of the system and the formula to be verified, we defined a structure to represent them. We also described the auxiliary functions that calculates the closure of a formula, the model checking nodes, the model checking graph and the strongly connected components (SCC).

Source code and documentation of the prototype can be found on <http://escher.puj.edu/~jearias/files/tccModelChecking.zip>.

Concluding Remarks

We conclude this document by stating the main results derived from this degree project and we also identify some directions for future work.

8.1 Overview

In this degree project we studied Model Checking as a formal method for the verification of `tcc` programs. To do this, we developed a model checking algorithm for `tcc`. The method proposed is based on the work by Falaschi and Villanueva [FV06].

We defined a structure called `tcc` Structure which is able to model the behavior of a `tcc` system. We also described a procedure to construct this structure from a `tcc` specification. Since the construction rules of the model follows the operational semantics of `tcc`, the resulting graph will consist of many node (state explosion problem). We addressed this problem by introducing a method to simplify the graph by removing its internal transitions. We illustrated the construction process and we obtained a finite-state model from a system that runs forever (Chapter 4).

We also studied a temporal logic that allows to specify properties of `tcc` systems (Chapter 5). We used this logic because it is based on sequences of constraints instead of classical states. In order to specify properties, we made use of the temporal logic proposed in [Val05] to specify properties of `ntcc`. Since `tcc` is a subcalculus of `ntcc`, this logic worked for our purposes.

We introduced the algorithm that proves if the model of a system satisfies a formula (Chapter 6). Our algorithm is based on the classical algorithm of model checking for LTL. Thus, we presented a structure which combines the model of the system and the formula to be verified. Then, we specified the properties that must have this graph to determine if the formula is satisfied by the model.

Finally, we described a prototype of the model checking algorithm (Chapter 7). We illustrated its performance verifying a property of a simple system.

8.2 Future Work

The following are, in the author's opinion, some interesting directions for future work:

Local Operator. In chapter 4 we presented an algorithm to construct the model of a system from the `tcc` specification. However, the algorithm does not support the local operator. This restricts the model checking algorithm to certain types of programs and then, many interesting programs

can not be verified. An approach to handle this operator is to keep track of the “already used” (hidden) variables. Then, we require to augment the `tcc` structure with that set of variables.

Model Implementation. So far, the construction of the model is performed manually. This task takes a long time and it is prone to errors. For this reason, we believe that the development of a tool to perform this task automatically will make our technique more amenable for non-expert users. Furthermore, the phase of translation of the specification into the `tcc` Structure will be more reliable.

State Explosion Problem. The state explosion problem is inherent in the model checking technique. We attempted to mitigate this problem by reducing the number of states with our method. But even so, the number of states generated in the model checking graph is huge. An interesting work of research is to consider symbolic and abstract techniques in order to reduce the number of states of the system.

ntcc Model Checking. The `ntcc` [Val02] calculus is an extension of `tcc`. This calculus is founded upon solid mathematical principles and it has attained a wide range of applications in emergent areas such as security, system biology and multimedia interaction. In spite of its modeling success, at present, `ntcc` does not provide tools for the automatic verification of system properties. We strongly believe that our algorithm can be extended to verify properties in `ntcc`. Fundamentally, the model of the system must be adapted to be able to represent the non-deterministic computation.

Prototype Improvement. The current prototype is very simple and it is not equipped to perform certain operations. For example, the implementation has a very simple constraint system, and then, we only can verify basic programs. Thus, we think that the implementation can be enhanced with an improved constraint engine. Moreover, a better support for formulas can be provided in order to, for instance, compute automatically the negated form of the formulas in the verification phase.

Bibliography

- [Bae05] J.C.M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Science*, 335(2-3):131–146, May 2005. 5
- [BG91] Albert Benveniste and Paul Le Guernic. Synchronous Programming with Events and Relations: the SIGNAL Language and its Semantics. *Science of Computer Programming*, 16(2):103–149, 1991. 6
- [BG92] Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992. 1, 6
- [BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985. 5
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. 7, 10, 11, 12
- [BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990. 5
- [CE82] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lectures Notes in Computer Science*, pages 52–71, London, 1982. Springer-Verlag. 2, 11
- [CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, 1999. 2, 8, 33, 41
- [FL79] Michael J. Fischer and Richard E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979. 33
- [Flo67] R. W. Floyd. Assigning Meaning to Programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. American Mathematical Society, 1967. 2
- [FV06] Moreno Falaschi and Alicia Villanueva. Automatic Verification of Timed Concurrent Constraint Programs. *Theory and Practice of Logic Programming*, 6(3):265–300, 2006. 1, 2, 3, 17, 61
- [Gar08] Hubert Garavel. Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular. In *Proceedings of LIX Colloquium on Emergent Trends in Concurrency Theory*, volume 209 of *Electronic Notes in Theoretical Computer Science*, pages 149–164. Elsevier, 2008. 2

- [HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. 6
- [Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969. 2
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. 1, 5
- [HR00] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, New York, 2000. 9
- [LP85] Orna Lichtenstein and Amir Pnueli. Checking that Finite State Concurrent Programs Satisfy their Linear Specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 97–107, New York, NY, USA, 1985. ACM Press. 33, 41
- [Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. 1, 5
- [Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. 1
- [MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992. 30
- [MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., 1995. 17, 33, 41
- [NPV02] Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. Temporal Concurrent Constraint Programming: Denotation, Logic and Applications. *Nordic Journal of Computing*, 9(2):145–188, 2002. 13
- [NV03] Mogens Nielsen and Frank D. Valencia. Notes on Timed CCP. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 702–741. Springer-Verlag, 2003. 22
- [Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, 1977. IEEE Computer Society. 9
- [QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the Fifth Colloquium on International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, London, 1982. Springer-Verlag. 2, 11
- [Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993. 1, 5, 13, 16

- [SJG94a] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of Timed Concurrent Constraint Programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80. IEEE Computer Press, 1994. 1, 6, 13, 14, 15, 16
- [SJG94b] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Programming in Timed Concurrent Constraint Languages. In *Constraint Programming: Proceedings 1993*, NATO Advanced Science Institute Series, pages 361–410, Berlin, 1994. Springer-Verlag. 1, 6, 13, 14, 15, 16
- [Smo94] Gert Smolka. A Foundation for Higher-order Concurrent Constraint Programming. In *Proceedings of the First International Conference on Constraints in Computational Logics*, pages 50–72, London, 1994. Springer-Verlag. 13
- [SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–352, New York, 1991. ACM Press. 1, 5, 13, 16
- [SW03] Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003. 1
- [Val02] Frank D. Valencia. *Temporal Concurrent Constraint Programming*. Phd thesis, University of Aarhus, 2002. 29, 62
- [Val05] Frank D. Valencia. Decidability of Infinite-State Timed CCP Processes and First-Order LTL. *Theoretical Computer Science*, 330(3):577–607, 2005. 29, 61