Predicate Diagrams as Basis for the Verification of Reactive Systems

Cecilia E. Nugraheni

München, 2004

Predicate diagrams as Basis for the Verification of Reactive Systems

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Cecilia E. Nugraheni
aus Surakarta, Indonesien

München, 9. Januar 2004

Erstgutachter: Prof. Dr. Fred Kröger
Zweitgutachter: Dr. Stephan Merz
Tag der mündlichen Prüfung: 13. Februar 2004

Abstract

This thesis proposes a diagram-based formalism for verifying temporal properties of reactive systems. Diagrams integrate deductive and algorithmic verification techniques for the verification of finite and infinite-state systems, thus combining the expressive power and flexibility of deduction with the automation provided by algorithmic methods.

Our formal framework for the specification and verification of reactive systems includes the Generalized Temporal Logic of Actions (TLA*) from Merz for both mathematical modeling reactive systems and specifying temporal properties to be verified. As verification method we adopt a class of diagrams, the so-called *predicate diagrams* from Cansell et al.

We show that the concept of predicate diagrams can be used to verify not only *discrete systems*, but also some more complex classes of reactive systems such as *real-time systems* and *parameterized systems*. We define two variants of predicate diagrams, namely *timed predicate diagrams* and *parameterized predicate diagrams*, which can be used to verify real-time and parameterized systems.

We prove the completeness of predicate diagrams and study an approach for the generation of predicate diagrams. We develop prototype tools that can be used for supporting the generation of diagrams semi-automatically.

Zusammenfassung

In dieser Arbeit schlagen wir einen diagramm-basierten Formalismus für die Verifikation reaktiver Systeme vor. Diagramme integrieren die deduktiven und algorithmischen Techniken zur Verifikation endlicher und unendlicher Systeme, dadurch kombinieren sie die Ausdrucksstärke und die Flexibilität von Deduktion mit der von algoritmischen Methoden unterstützten Automatisierung.

Unser Ansatz für Spezifikation und Verifikation reaktiver Systeme schließt die *Generalized Temporal Logic of Actions* (TLA*) von Merz ein, die für die mathematische Modellierung sowohl reaktiver Systeme als auch ihrer Eigenschaften benutzt wird. Als Methode zur Verifikation wenden wir Prädikatendiagramme von Cansell et al. an.

Wir zeigen, daß das Konzept von Prädikatendiagrammen verwendet werden kann, um nicht nur diskrete Systeme zu verifizieren, sondern auch kompliziertere Klassen von reaktiven Systemen wie Realzeitsysteme und parametrisierte Systeme. Wir definieren zwei Varianten von Prädikatendiagrammen, nämlich gezeitete Prädikatendiagramme und parametrisierte Prädikatendiagramme, die benutzt werden können, um die Realzeit- und parametrisierten Systeme zu verifizieren.

Die Vollständigkeit der Prädikatendiagramme wird nachgewiesen und ein Ansatz für die Generierung von Prädikatendiagrammen wird studiert. Wir entwickeln prototypische Werkzeuge, die die semi-automatische Generierung von Diagrammen unterstützen.

Contents

List of Figures

Chapter 1

Introduction

The dependency on its own technological achievements by modern society is getting more and more. Powerful computer systems, which are the backbones of almost conceivable technology today, are in the development or in the implementation stages. The complexity of these systems is growing ceaselessly. Most of today's computing systems are characterized by an ongoing interaction with their environments. This interaction occurs in various forms such as the transmission of data over a communication network to another machine, interaction with a human user, or the exchange of information with the sensors and actuators of an embedded control system. Such systems are called *reactive*, in contrast to *transformational* systems that compute an output from a given input.

Considering our dependency on these systems, it is clear that they should be correct. Reactive systems are most often composed of several communicating concurrent processes. The inherent complexity of concurrency and communication makes the discovery of design errors a difficult task. Not only may there be mistakes in the *calculations* such system perform (as in transformational systems), but there is also the possibility of *synchronization* failures (such as deadlocks, starvation, unexpected message reception etc.). One of the most challenging problems facing today's software engineers and computer scientists is therefore to find ways and establish techniques in order to reduce the number of errors in reactive systems.

This thesis presents a methodology for the formal analysis of some classes of reactive systems.

1.1 Classification of reactive systems

Reactive systems are commonly classified as *discrete, real-time* and *hybrid* [77]:

- A discrete system only represents the qualitative aspect of time, that is the order of events, but does not measure the time elapsed between these events. The behavior is fully described by the discrete events.
- A real-time system captures the metric aspects of time; discrete events may have time stamps.
- In hybrid systems, we allow the inclusion of variables that evolve continuously over time between discrete events. The evolution of the continuous variables is described separately from the discrete events, usually by differential equations.

The behavior of a reactive system can be characterized in different ways, for example by the stream of outputs produced by the system, or by the actions taken by the system. In this thesis, a reactive system is characterized by the sequence of states, which are interpretations of the variables traversed by the system. We call a system *finite state* if this set is finite, and otherwise *infinite state*. Of course all real-time and hybrid systems are infinite-state due to the presence of real-valued clock and continuous variables.

Reactive systems usually consist of a collection of processes running parallel in the systems. *Parallel systems* can be classified as *interleaving* and *non-interleaving* systems. An interleaving system is a system in which each step can be attributed to exactly one process. A non-interleaving system allows steps that represent simultaneous operations of two or more different processes. When a parallel system consists of a collection of identical processes, it is categorized as a *parameterized system*.

1.2 Formal specification and verification

Formal methods are a collection of notations and techniques for describing and analyzing systems. These methods are *formal* in the sense that they are based on some mathematical theories, such as logic, automata or graph theory. They are aimed at enhancing the quality of systems. Formal specification techniques introduce a precise and unambiguous description of the properties of systems. This is useful in eliminating misunderstanding and can be used further for debugging systems. Formal analysis techniques can

be used to verify that a system satisfies its specification or to systematically seek for cases where it fails to do so.

A formal framework for the specification and verification of reactive systems should include at least the following parts [78, 88]:

- a *mathematical model* of reactive systems,
- a *requirement specification* or *property* languages and
- a *verification method*.

Model The large majority of frameworks that include a verification method use transition systems as their computational model of reactive systems. This is certainly due to the simplicity of this model. A transition system is essentially a graph, where the nodes represent system's states and the edges represent the atomic transitions between these states. Concurrency is modeled by non-deterministic interleaving of atomic actions. Another important ingredient in the description of transition systems for modeling reactive systems are fairness and liveness conditions that require some actions to be eventually taken or some state to be eventually reached. These conditions help to balance the local non-determinism of transition systems of choosing among actions that are permitted at a given source state and lead to different possible target states.

Property In order to reason about the behavior of reactive systems, *temporal logic* was proposed as a convenient specification or property language. Temporal logics are extensions of classical (propositional and/or first-order) logics, incorporating a model of the flow of time, either as metric constraints or via a suitable semantics. Temporal logics are often classified according to whether time is assumed to have a *linear* or a *branching* structure.

Verification Verification of reactive systems consists of establishing whether a reactive systems satisfies its specification, that is, whether all possible behaviors of the system are included in the property specified. For finite-state systems and a restricted class of infinite-state systems, verification of temporal-logic properties is decidable: algorithms can be devised that determine in a finite number of steps whether a system satisfies its specification. For the vast majority of infinite-state systems no such algorithms exist; the problem is undecidable. In this case, verification relies on human interaction and heuristics.

The use of temporal logics for the specification and verification of reactive systems goes back to Pnueli's seminal paper on temporal logic [94]. Formulas of temporal logic are interpreted over runs of transition systems and can thus express properties of reactive systems. Many useful properties of reactive systems can be expressed in temporal logics, including safety properties ("nothing bad happens") and liveness properties ("something good happens").

1.3 Verification techniques

There are basically two approaches to verification of reactive systems: the algorithmic approach on one hand and the deductive approach on the other hand. When verifying temporal properties of reactive systems, algorithm methods are used when the problem is decidable and deductive methods are employed otherwise.

The most popular algorithmic verification method is *model checking*, independently proposed by Clarke & Emerson [27, 28] and Queille & Sifakis [95]. A complete state graph of the system is built and specialized methods are used to check whether all paths through this graph conform to some properties. A *counterexample* is found whenever a path that does not satisfy a temporal property is encountered. Although this method is fully automatic for finite-state systems, it suffers from the so-called *state-explosion* problem. The size of the state space is typically exponential in the number of components, and therefore the class of systems that can be handled by this method is limited. State space reduction techniques such as symbolic representations [25, 82, 22], symmetry [29, 43] and partial order reductions [52, 92, 105] have yielded good results but the state spaces that can be handled in this manner are still quite modest.

Automata-theoretic verification methods [108, 66] are closely related to model checking, in which both the system and the property are represented by ω-automata (automata on infinite words) and automata-theoretic methods are used to establish language inclusion.

On the other end of the spectrum we find deductive verification methods based on theorem proving; these methods are extension of the proof methods originally established by Floyd [47] and Hoare [56] for sequential systems. They typically reduce the proof of a temporal property to a set of proofs of first-order verification conditions, which can then be dealt by standard theorem provers. Deductive methods are very powerful and generally applicable to infinite-state systems, but suffer from the high level of user interaction

required to complete a proof.

While it is clear that any way out of this impasse must rely on a combination of theorem proving and model checking, specific methodologies are needed to make such a combination work with a reasonable degree of automation.

1.4 Abstraction

An attractive method for proving a temporal property φ for a reactive system $\mathcal{S}$ is to find a simpler abstract system $\mathcal{A}$ such that if $\mathcal{A}$ satisfies φ then $\mathcal{S}$ satisfies φ as well. In particular, if $\mathcal{A}$ is finite-state, the validity of φ for $\mathcal{A}$ can be established automatically using a model checker, which may not have been possible for $\mathcal{S}$ due to an infinite or overly large state-space.

Thus, abstraction is a key methodology in combining deductive and algorithmic techniques. Abstraction can be used to reduce problems to model-checkable form, where deductive tools are used to construct valid abstract descriptions or to justify that a given abstraction is valid.

There is much work on the theoretical foundations of reactive system abstraction [31, 37, 74, 36, 53, 33, 75, 40], usually based on the ideas of abstract interpretation [34].

Most abstractions weakly preserve temporal properties: if a property holds for the abstract systems, then a corresponding property will hold for the concrete one. However, the converse will not be true: not all properties satisfied by the concrete system will hold at the abstract level. Thus, only positive results transfer from the abstract to the concrete level. This means, in particular, that abstract counter-examples will not always correspond to concrete ones.

1.5 Diagram-based verification

The deductive approach for verifying temporal properties of reactive systems is based on *verification rules*, which reduce the system validity of a temporal property to the general validity of a set of first-order *verification conditions*. While this methodology is complete, relative to the underlying first-order reasoning, the proofs do not always reflect an intuitive understanding of the system and its specification; without this intuition, the proofs can be difficult to construct.

The need for a more intuitive approach to verification leads to the use of *diagram-based formalisms*. Usually, these diagrams are graphs whose vertices

are labeled with first-order formulas, representing sets of system states, and whose edges represent possible system transitions. This approach combines some of the advantages of deductive and algorithmic verification: the process is goal-directed and incremental, and can handle infinite-state systems.

Some features shared by these formalisms are [19, 26]:

- Diagrams (or sequences of diagrams) are formal proof objects, which succinctly represent a set of verification conditions that replaces a combination of textual verification rules.
- The graphical nature of diagrams makes them easier to construct and understand than text-based proofs and specifications.
- Diagrams can describe and verify infinite-state systems using a finite and often compact representation.
- Diagrams can be viewed as the abstract representation of the systems being considered.
- The construction of a diagram can be incremental, starting from a high-level outline and then filling in details as necessary.
- The verification conditions are *local* to the diagram; failed verification conditions point to missing edges or nodes, or possible bugs in the system. The necessary global properties of diagrams can be proved algorithmically.
- Besides their use as a formal basis for verification, diagrams can also serve as support for explaining how systems are working and for documenting them.

There are some work using graphs to visualize and structure temporal proof, for example the diagram from Owicki and Lamport [90], *proof charts* from Cousot [35], *Predicate-action diagrams* proposed by Lamport [70], *verification diagrams* from Manna & Pnueli [79], *generalized verification diagram* from Sipma [99] and *predicate diagrams* from Cansell et al. [26].

1.6 Scope of the thesis

In this work, three classes of reactive systems are considered: discrete systems, real-time systems and parameterized systems. We will use the Generalized Temporal Logic of Action (TLA*) from Merz [83], which is a variant

of Temporal Logic of Action (TLA) from LAMPORT [69], to formalize our methodology. Our choice is due to its completeness; in the sense that it provides all the components of a formal framework for the specification and verification of reactive systems as mentioned in Section 1.2. Like in TLA, in TLA* there is no distinction between systems and properties, both are represented as formulas. It also provides proof systems that can be used to prove that a specification satisfies a desired property. This verification process is reduced to the proof that the specification implies the property.

In this thesis, we will follow the diagram-based verification techniques. The verification will be done by means of predicate diagrams from CANSELL et al. It is already shown that this diagram is suitable to TLA formalism [26].

The main goal of this thesis can be stated in the following questions:

1. How can reactive systems be represented in TLA*?
2. How can predicate diagrams be used to verify discrete systems?
3. What about the completeness of predicate diagrams, i.e. for any specification and any formula of the temporal propositional logic, if the specification implies the formula, can the implication be proven by a suitable predicate diagram?
4. How far can predicate diagrams be used to verify some other classes of reactive systems, in particular the more complex systems than discrete systems such as real-time systems and parameterized systems?
5. Is it possible to generate or to construct predicate diagrams automatically?

1.7 Chapter outlined

In Chapter 2 mathematical preliminaries are introduced, including set notations, strings and languages, graphs and classical logics, in order to establish the terminology and notational used in this book.

Chapter 3 is addressed to the properties of reactive systems. First, we give the very general definition of properties as arbitrary subsets of infinite states. Second, we give the definition of the syntax and semantics of TLA* introduced by MERZ [83]. We also present the general form of specification we will use in this sequel and introduce the writing style for writing specifications in the next sections.

In the following chapter, Chapter 4, we will consider a class of finite automata over infinite words, called Muller automata [86], including the formal definition and an algorithm for translating pTLA* formulas to a Muller automaton. We also briefly describe timed automata, which will be used in the verification of real-time systems.

Chapter 5 deals with the verification of discrete systems. We define the formula that will be used to represent the specification of discrete systems. We then present predicate diagrams, including the definition and the steps to be done in order to verify discrete systems using the diagrams. As illustration, we take the Bakery Algorithm. We show that predicate diagram is complete, i.e. for any specification and any formula of the temporal propositional logic, if the specification implies the formula, then there exists a suitable predicate diagram that can be used to prove the implication.

Chapter 6 is devoted to the specification and verification of real-time systems. First, we give the standard formula for real-time specification we use in this book. Second, we define a variant of predicate diagrams, which we call *timed predicate diagrams* that can be used to verify real-time systems. As illustration, we take the FISCHER's protocol problem.

The verification of parameterized systems will be considered in the following chapter. After defining the specification of parameterized systems, we explain how can predicate diagrams be used to verify the properties related to whole processes in the protocol. As a motivated example we take the Tickets protocol. We then define a variant of predicate diagrams called *parameterized predicate diagrams* that can be used to verify the property of a single process in the Tickets protocol.

In Chapter 8 the method for automatically generation of diagrams will be studied. It is started by briefly describing the concept of abstract interpretation and the algorithm of the diagrams generation. Two tools that have been developed in this work will be presented. Then it will be shown how these tools can be used in the generation of diagrams for the case studies presented in the previous chapters, namely the Bakery algorithm and the Ticket protocol.

The final chapter concludes the thesis with a review of its goals and their achievements and an outlook on future research.

Chapter 2

Preliminaries

2.1 Overview

This chapter is devoted to mathematical preliminaries, including set notations, strings and languages, graphs and classical logics. The objective of this chapter is to establish the terminology and notational used in this book. For more details, the reader may consult Gallier [48], Peled [93], Kröger [65] and Fitting [46].

2.2 Set Notation

A *set* is a finite or infinite collection of elements. Repetitions of elements in a set are ignored. For finite sets, the elements can be listed between a matching pair of braces, as in $\{1, 3, 5\}$. Another notation for sets is $\{x : R(x)\}$, where R is some description that restricts the possible values of x. In the sequel, $\mathbb{R}$ denotes the set of real numbers, $\mathbb{R}^+$ denotes the set of nonnegative real numbers, $\mathbb{N}$ denotes the set of natural numbers and ω denotes the smallest infinite ordinal number where $n < \omega$ holds for every $n \in \mathbb{N}$.

One special set is the *empty set*, denoted by $\emptyset$, which does not contain any elements. The *size* of a set is the number of elements it contains. For a finite set A, the size of A is denoted by $|A|$. Obviously, $|\emptyset| = 0$.

To denote that an element x belongs to a set A, we write $x \in A$. If x does not belong to A, we write $x \notin A$. One can compare a pair of sets as follows:

- $A \subseteq B$ if for every element $x \in A$ it is also the case that $x \in B$. We say that A is *contained* in B or that A is a *subset* of B. In this case we can also write $B \supseteq A$.

- $A = B$ if both A is contained in B and B is contained in A. We say that A and B are *equal*.
- $A \subset B$ if A is contained in B but A is not equal to B (thus, B must include at least one element that is not in A). We say that A is *properly contained* in B or that A is *a proper subset* of B. We can also write $B \supset A$.

There are some usual operations that can be performed on sets:

- $A \cup B$, the *union* of A and B, is $\{x : x \text{ is in } A \text{ or } x \text{ is in } B\}$.
- $A \cap B$, the *intersection* of A and B, is $\{x : x \text{ is in } A \text{ and } x \text{ is in } B\}$.
- $A \setminus B$, the *difference* of A and B, is $\{x : x \text{ is in } A \text{ and } x \text{ is not in } B\}$.
- $A \times B$, the *cartesian product* of A and B, is the set of *ordered pairs* (x_1, x_2) such that x_1 is in A and x_2 is in B.
- 2^A, the *power set* of A, is the set of all subsets of A.

A *binary relation* between A and B is any subset R (possibly empty) of $A \times B$. Given a relation R between A and B, the set

$$\{x_1 \in A : \text{there exists } x_2 \in B \text{ such that } (x_1, x_2) \in R\},$$

is called the *domain* of R and is denoted by $dom(R)$. The set

$$\{x_2 \in B : \text{there exists } x_1 \in A \text{ such that } (x_1, x_2) \in R\}$$

is called the *range* of R and is denoted by $range(R)$. When $A = B$, a relation R between A and A is also called a relation *on* (or *over*) A. We will also use the notation $x_1 R x_2$ as an alternate to $(x_1, x_2) \in R$. We say a relation R on set S to have the property listed in the left-hand column of Table 2.1 iff the corresponding condition in the right-hand column is satisfied for all x_1, x_2, x_3 in S.

A relation R is an *equivalence relation* if it is reflexive, symmetric and transitive.

A relation R on a set A is a *partial order* iff R is reflexive, transitive and antisymmetric. A partial order is often denoted by the symbol $\succeq$.

Given a partial order $\succeq$ on a set A, given any subset X of A, X is a *chain* iff for all $x, y \in X$, either $x \succeq y$ or $y \succeq x$. The *strict order* $\succ$ associated with $\succeq$ is defined as follows: $x_1 \succ x_2$ iff $x_1 \succeq x_2$ and $x_1 \neq x_2$.

Property	Condition: For all x_1, x_2, x_3 in S
1. Reflexivity	$x_1 R x_1$.
2. Irreflexivity	$x_1 R x_1$ is false for all x_1 in S.
3. Transitivity	$x_1 R x_2$ and $x_2 R x_3$ imply $x_1 R x_3$.
4. Symmetry	$x_1 R x_2$ implies $x_2 R x_1$.
5. Antisymmetry	If $x_1 R x_2$ and $x_2 R x_1$ then $x_1 = x_2$.

Table 2.1: Properties of binary relations.

A binary relation $\succeq$ is called *well-founded* on set S if there are no infinite descending chains, that is, no infinite sequences of elements $x_1, x_2, \ldots, x_n, \ldots$ in S such that $x_1 \succeq x_2 \succeq \ldots \succeq x_n \succeq \ldots$.

The *reflexive closure* of a binary relation R on a set S is the minimal reflexive relation on S that contains R. Thus $x_1 R' x_1$ for every element x_1 of S and $x_1 R' x_2$ for distinct elements x_1 and x_2, provided that $x_2 R' x_1$.

Let A and B be non-empty sets. A *function* (or *mapping*) f from A into B is a binary relation between A and B having the special property that if $x_1 f x_2$ and $x_1 f x_3$ then $x_2 = x_3$. One usually writes $x_2 = f(x_1)$, instead of $x_1 f x_2$. The element $f(x_1)$ of B is called the value of f at x_1 and the association that defines f can be written $f : x_1 \rightarrow f(x_1)$ and can be read f maps x_1 to $f(x_1)$.

A *relation* of arity n is a set of n-tuples over some domain. The fact that $(x_1, \ldots, x_n) \in R$ is often denoted by $R(x_1, \ldots, x_n)$. A *function* (or a *mapping*) of arity n can be viewed as a constrained relation, containing $(n+1)$-tuples, where the first n elements uniquely define the $(n+1)$st element. That is, one cannot have two $(n+1)$-tuple that agree on their first n elements but differ in their $(n+1)$st element. A function f over the domains $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$ that results in a value from the domain $\mathcal{D}_{n+1}$ will be denoted by $f : \mathcal{D}_1 \times \ldots \times \mathcal{D}_n \rightarrow \mathcal{D}_{n+1}$.

Given two sets I and X, an *I-indexed sequence* (or sequence) is any function $A : I \rightarrow X$, usually denoted by $(A_i)_{i \in I}$. The set I is called the *index set*. If X is a set of sets, $(A_i)_{i \in I}$ is called a *family* of sets.

2.3 Strings and languages

A *string* is a (finite or infinite) sequence over some predefined finite set called an *alphabet*. A (finite or infinite) set $\mathcal{L}$ of strings over some alphabet is called a *language*. Since a language is a set (of strings), it can be defined by using

the usual set notation.

There are several useful operations on strings that will be used in the sequel:

- Concatenation: connecting two or more strings together in some order; usually written in the required order and separated with '.' or '$\circ$'. For example, three string ab,ca,and dc can be concatenated, $ab.ca.dc$, resulting a new string $abcadc$.

- Set operators such as $\cup, \cap$, or $\setminus$ may be used to form a language from other given languages. Since languages are sets of strings, the set comparison relations $\subseteq, =$ and $\supseteq$ can also be used to compare between them.

Given a string u, a string v is a *prefix* of u if there is a string w such that $u = vw$. A string v is a *suffix* of u if there is a string w such that $u = wv$. A string v is a *sub-string* of u if there are strings x and y such that $u = xvy$.

Let Σ be a non-empty set. Σ^* and Σ^ω denote the sets of finite and infinite sequences, respectively, of elements of Σ (Σ-sequences, for short). A special string, denoted by ε (where ε is not in the alphabet Σ), is the *empty* string, containing no letters. $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ denotes the set of non-empty finite Σ-sequences. Let $\sigma = s_0 s_1 \ldots \in \Sigma^* \cup \Sigma^\omega$ be a Σ-sequence and $i \in \mathbb{N}$, then

- for $0 \leq i < |\sigma|, \sigma[i]$ denotes the i-th element s_i of σ,

- for $0 \leq i < |\sigma|, \sigma[..i]$ denotes the finite sequence $s_0 s_1, \ldots, s_i \in \Sigma^+$ (the prefix of σ up to and including s_i) and

- for $0 \leq i < |\sigma|, \sigma[i..]$ denotes the sequence $s_i, s_{i+1}, \ldots \in \Sigma^* \cup \Sigma^\omega$ (the suffix of σ starting at s_i).

2.4 Graphs

Let V be a finite, non-empty set and E be a binary relation on V. Then $G = (V, E)$ is called a *directed graph*, or *digraph*. An element of V is called a *vertex* or *node* and an element of E is called an *edge*. If $e = (n_1, n_2)$ is an edge in E, then n_1 is called the *source* of e and n_2 is called the *target* of e. One may also say that e is an *outgoing edge* of n_1 and an *ingoing edge* of n_2. The number of ingoing and outgoing edges of a node n is called *in-degree* and *out-degree* of n, respectively.

Digraphs are usually depicted using diagrams like one in Figure 2.1 [93]. In this graph, E is the binary relation $\{(n_1, n_2), (n_2, n_3), (n_2, n_5), (n_3, n_1),$

(n_3, n_4), (n_4, n_4), (n_4, n_8), (n_5, n_6), (n_5, n_9), (n_6, n_7), (n_7, n_8), (n_8, n_5), $(n_9, n_7)\}$ over the set $\{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$. In such a diagram, nodes are shown as circles and edges are represented as arrows stretched between the circles representing related nodes.

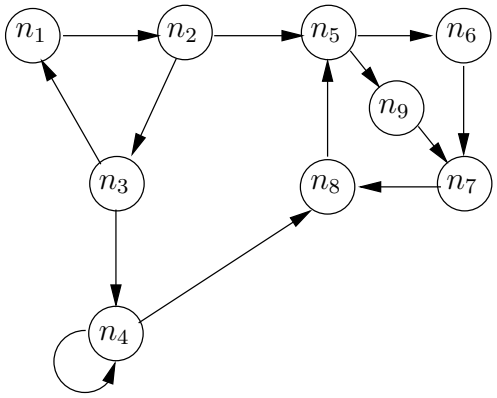

Figure 2.1: A directed graph.

An edge such as (n_4, n_4) is called an *self-loop*. A digraph with no self-loops is called *loop-free*. Thus, a digraph (V, E) is loop-free iff E is a irreflexive relation.

If the relation E is symmetric, i.e. if for each $(s, r) \in E$ we also have $(r, s) \in E$, then the graph is called *undirected* and the edges are usually represented by lines, instead of arrows, connecting two related nodes.

Some additional components may be included in a graph. A graph may contain *labels* on the nodes, the edges or both. Given a set of labels L, an *edge labeling function* $\ell : E \to L$ assigns to each edge in E an element of L. In this case, the graph is a quadruple (V, E, L, ℓ). Labels can allow us to distinguish between edges when there is more than one edge between certain pairs of nodes. Then the edges can also be redefined as triples over $V \times L \times V$, so that each edge contains the source node, the label and the target node. In this case, the edge labeling function ℓ for an edge (s, a, r) returns the projection on the middle component a of the edge (s, r).

A *path* is a (finite or infinite) sequence of nodes of V, $n_0, n_1, \ldots, n_k, \ldots$ such that each adjacent pair of nodes n_i, n_{i+1} forms an edge $(n_i, n_{i+1}) \in E$. A path is *simple* if no node on it appears more than once. Notice that in an infinite path over a finite graph, at least one of the nodes must repeat infinitely many times. A *cycle* is a finite path that begins and ends with the

same node. The *length* of a path is the number of edges that appear on it, including repetitions (hence, the number of nodes on a path is one more than the length of the path). Consequently, for each node there is a trivial path from the node to itself of length 0. In the graph of Figure 2.1, there is a simple path $n_1, n_2, n_3, n_4, n_8, n_5, n_9, n_7$ of length 7. The path n_5, n_6, n_7, n_8, n_5 is a cycle, whose length is 4.

A subset of nodes $V' \subseteq V$ in a graph is called a *strongly-connected subgraph* (SCS) if there is a path between any pair of nodes in V', passing only through nodes in V'. A *strongly connected component* (SCC) or *maximal strongly connected subgraph* (MSCS) is a maximal set of such nodes, i.e. one cannot add any node to that set of nodes and still maintain strong connectivity. A *trivial* SCC is a SCC consisting of one node without a self-loop. The graph in Figure 2.1 has five SCSs: $\{n_1, n_2, n_3\}, \{n_4\}$, $\{n_5, n_6, n_7, n_8\}$, $\{n_5, n_7, n_8, n_9\}$, and $\{n_5, n_6, n_7, n_8, n_9\}$ and three of them are SCCs, namely $\{n_1, n_2, n_3\}, \{n_4\}$ and $\{n_5, n_6, n_7, n_8, n_9\}$.

A *rooted graph* is an arbitrary graph with one of its nodes is labeled in a special way so as to distinguish it from other nodes. The special node is called the *root* of the graph. A *tree* is a cycle-free rooted digraph whose set of nodes is not empty. The root of a tree has in-degree 0 and every node other than the root has in-degree 1. For every node n of a tree there is a path from the root to n. If there is a path from a node n_1 to a node n_2 in a tree, then n_2 is called a *successor* of n_1 and n_1 is a *predecessor* of n_2.

2.5 Classical logic

There are many useful logics that differ in what concepts are being considered and in what the basic features of these concepts are thought to be. In the family of formal logics, one is central: classical logic. It is the most widely used logic, the logic underlying mathematics as it is generally practiced and on the top of which many others have been built.

In the following we present some concepts and notions of classical logic including *propositional* and *first order logic*.

2.5.1 Propositional logic

A *logical language* is given by an alphabet of symbols and the definition of a set of strings over this alphabet called *formulas*. The simplest kind of such a language is a language $\mathcal{L}_p$ of *(classical) propositional logic* that can be given as follows.

2.5.1.1 Syntax

Alphabet
The alphabet of $\mathcal{L}_p$ consists of a constant symbol **false**, a countable set of propositional letters $\mathcal{V}$ and symbols $\rightarrow$, (and).

Atomic formulas
Every atomic proposition $v \in \mathcal{V}$ and **false** is an *atomic formula*.

Formulas
The inductive definition of formulas is given as follows.

1. Every atomic formula is a formula.
2. If F and G are formulas then $(F \rightarrow G)$ is a formula.

Abbreviations
Further logical operators and a constant can be introduced to abbreviate particular formulas:

- $\neg F$ for $F \rightarrow$ **false**,
- $F \vee G$ for $(\neg F) \rightarrow G$,
- $F \wedge G$ for $\neg(\neg F \vee \neg G)$,
- $F \leftrightarrow G$ for $(F \rightarrow G) \wedge (G \rightarrow F)$ and
- **true** for $\neg$**false**.

Notice that we omit surrounding parentheses.

Sub-formulas
Occasionally, we will need the notion of *sub-formulas*. Informally, a sub-formula of a formula is a substring that, itself, is a formula.

Definition 2.1 *(immediate sub-formula) Immediate sub-formulas are defined as follows:*

1. *An atomic formula has no immediate sub-formulas.*
2. *The immediate sub-formulas of* $(F \rightarrow G)$ *are* F *and* G*.*

Definition 2.2 *(sub-formula of $\mathcal{L}_p$) Let F be a formula. The set of sub-formulas of F is the smallest set S that contains F and contains, with each member, the immediate sub-formulas of that member. F is called an improper sub-formula of itself.*

2.5.1.2 Semantics

The *semantics* of such a language $\mathcal{L}_p$ is based on the concept of (*Boolean*) *valuation*, which is a mapping

$$s : \mathcal{V} \to \{\mathsf{tt}, \mathsf{ff}\}$$

where tt and ff are called truth values (representing "true" and "false", respectively). Every s can be deductively extended to the set of all formulas:

- $s(\mathbf{false}) = \mathsf{ff}$
- $s(v)$ for $v \in \mathcal{V}$ is given
- $s(F \to G) = \mathsf{tt}$ iff $s(F) = \mathsf{ff}$ or $s(G) = \mathsf{tt}$

Validity
A formula F is called *valid in* s (denoted by $\models_s F$) if $s(F) = \mathsf{tt}$. F is called *valid* or *tautology* (denoted by $\models F$) if $\models_s F$ holds for every s.

2.5.2 First order logic

Starting from some atomic formulas of which no further details are given, propositional logic investigates the logic operation such as $\neg, \to \vee, \wedge$, etc. We now present a logic based on propositional logic which additionally looks closer at the structure of atomic formulas and allows quantification. We call such logic *first-order predicate logic*.

A (*classical*) *first-order language* is given as follows.

2.5.2.1 Syntax

Alphabet
The alphabet of first order logic consists of the alphabet of $\mathcal{L}_p$, additional connection symbols ':', ',', and '=', quantifier symbol $\exists$ and a set V of variable symbols $x, y, z, \ldots$.

Functions and predicates
A first-order language is determined by specifying a finite set $\mathcal{F}$ of function symbols and a finite set $\mathcal{P}$ of predicate symbols. The first-order language determined by $\mathcal{F}$ and $\mathcal{P}$ is denoted by $\mathcal{L}(\mathcal{F}, \mathcal{P})$.

Terms
The family of terms of $\mathcal{L}(\mathcal{F}, \mathcal{P})$ is the smallest set meeting the conditions:

1. Any variable is a term.

2. If f is an n-ary function symbol (member of $\mathcal{F}$) and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

A term is *closed* if it contains no variables.

Atomic formulas
An atomic formula of $\mathcal{L}(\mathcal{F}, \mathcal{P})$ is **false** or any string of the form $P(t_1, \ldots, t_n)$ where P is an n-ary predicate symbols (member of $\mathcal{P}$) and $t_1, \ldots, t_n$ are terms of $\mathcal{L}(\mathcal{F}, \mathcal{P})$.

Formulas
The family of formulas of $\mathcal{L}(\mathcal{F}, \mathcal{P})$ is the smallest set meeting the following conditions:

1. Any atomic formula is a formula.

2. If t_1 and t_2 are terms then $t_1 = t_2$ is a formula.

3. If F and G are formulas so is $(F \to G)$.

4. If P is an n-ary predicate symbol (member of $\mathcal{P}$) and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a formula.

5. If F is a formula and x is a variable, then $\exists\, x \,:\, F$ is a formula.

Abbreviations
In addition to the abbreviations as in $\mathcal{L}_p$, the formula of the form $\forall\, x \,:\, F$ is introduced as the abbreviation of the formula $\neg(\exists\, x \,:\, \neg F)$.

Occurrence of variables and substitutions
The occurrence of a variable x in some formula F is called *bound* if it appears in some sub-formula $\exists\, x\, :\, G$, consequently in $\forall\, x\, :\, G$, of F. Otherwise it is called *free*.

If t is a term then $F[t/x]$ denotes the result of *substituting* t for every free occurrence of x in F. When writing $F[t/x]$ we always assume implicitly that t does not contain a variable which occur bound in F. (This can always be achieved by replacing the bound variables of F by others.)

2.5.2.2 Semantics

The basic semantical concept of first-order logic is the following:

Structure
A *structure* I for $\mathcal{L}(\mathcal{F}, \mathcal{P})$ consists of

1. a set $|I| \neq \emptyset$, called *universe*,
2. an n-ary function $I(f) : |I|^n \to |I|$ for every n-ary function symbol f in $\mathcal{F}$,
3. an n-ary relation $I(P) \subset |I|^n$ for every n-ary predicate symbol P in $\mathcal{P}$.

Variable valuation
A *variable valuation* ξ (with respect to I) is a mapping $\xi\, :\, V \to |I|$ which assigns some $\xi(x) \in |I|$ to every variable x of $\mathcal{L}(\mathcal{F}, \mathcal{P})$.

Term semantics
A structure I together with a variable valuation ξ defines a value $I^\xi(t) \in |I|$ for every term t:

1. $I^\xi(x) = \xi(x)$ for every variable x.
2. $I^\xi(f(t_1, \ldots, t_n)) = I(f)(I^\xi(t_1), \ldots, I^\xi(t_n))$.

Formula semantics
The semantics of formulas, relative to some interpretation I and some valuation $\xi\, :\, V \to |I|$ is inductively defined as follows:

1. $I^\xi(\mathbf{false}) = \mathsf{ff}$.
2. $I^\xi(t_1 = t_2) = \mathsf{tt}$ iff $I^\xi(t_1) = I^\xi(t_2)$, where $=$ denotes equality in $|I|$.

3. $I^{\xi}(F \to G) = \mathsf{tt}$ iff $I^{\xi}(F)$ implies $I^{\xi}(G)$.
4. $I^{\xi}(P(t_1, \ldots, t_n)) = \mathsf{tt}$ iff $(I^{\xi}(t_1), \ldots, (t_n)) \in I^{\xi}(P)$ for every P in $\mathcal{P}$.
5. $I^{\xi}(\exists\, x : F) = \mathsf{tt}$ iff $I^{\zeta}(F) = \mathsf{tt}$ for some valuation ζ such that $\xi(y) = \zeta(y)$ for all $y \neq x$.

Validity

A formula F of $\mathcal{L}(\mathcal{F}, \mathcal{P})$ is called *valid in* I (denoted by $\models_I F$) if $I^{\xi}(F) = \mathsf{tt}$ for every ξ. F is called *valid* or *tautology* (denoted by $\models F$) if $\models_I F$ holds for every I.

Chapter 3

Properties and Temporal Logic

3.1 Overview

This chapter is about the properties of reactive systems. Whereas in Section 3.2 we deal with a very general definition of properties as arbitrary subsets of infinite runs, in Section 3.3 we give the definition of the syntax and semantics of TLA* introduced by Merz [83]. We also present the general form of specification we will use in this sequel and describe briefly the issues on *machine-closed*. Then we introduce the writing style for writing specifications in the next sections. A short remark will be given in the end of this chapter.

3.2 Properties of reactive systems

Let Σ be a set (of states). Informally Σ may be thought of as the set of states a reactive system may assume. A *run* of the system can be represented as an infinite sequence $\sigma = s_0 s_1 \ldots \in \Sigma^\omega$. A $(\Sigma-)$*property* P is any set $P \subseteq \Sigma^\omega$ of runs.

To introduce some structure into the class of properties, two basic classes of properties are usually distinguished in the literature, namely *safety properties* and *liveness properties*.

3.2.1 Safety properties

Safety properties are formally defined as follows:

Definition 3.1 *(safety property) A property $P \subseteq \Sigma^\omega$ is a safety property iff for every $\sigma \in \Sigma^\omega$,*

$$\sigma \in P \longleftrightarrow \forall i \geq 0 \,:\, \exists \tau \in \Sigma^\omega \,:\, \sigma[..i]\tau \in P.$$

A safety property is true for an infinite behavior σ iff it is true for all finite prefixes of σ. Informally, P is a safety property if for every run σ *not* contained in P there is a finite prefix $\sigma[..i]$ which can not be complemented by any sequence $\tau \in \Sigma^\omega$ to obtain a run $\sigma[..i]\tau$ in P. Stated differently, for every $\sigma \notin P$ something "bad" must have happened after some finite number of steps which cannot be remedied by any future behavior. In LAMPORT's popular characterization, safety property express that "something bad never happens".

The class of safety properties is closed under intersection and union.

Lemma 3.2

- $\bigcap_{i \in I} P_i$ *is a safety property if all* P_i *(for* $i \in I$*) are safety properties.*
- *Let* $P_1, \ldots, P_n$ *be safety properties. Then* $\bigcup_{i=1}^{n} P_i$ *is a safety property.*

Notice that in general only finite unions of safety properties are themselves safety properties.

For every property P there is a safety property S such that $S \supseteq P$. By Lemma 3.2, there is a smallest safety property (w.r.t. set inclusion) containing P.

Definition 3.3 *(safety closure) Let* P *be any property. The safety closure of* P*, written safe(P), is the smallest safety property containing* P*,*

$$safe(P) = \bigcap \{S \subseteq \Sigma^\omega \,:\, S \text{ is a safety property and } P \subseteq S\}.$$

3.2.2 Liveness properties

Liveness properties are formally defined as follows:

Definition 3.4 *(liveness property) A property* $P \subseteq \Sigma^\omega$ *is a liveness property iff*

$$\forall \sigma \in \Sigma^* \,:\, \exists \tau \in \Sigma^\omega \,:\, \sigma\tau \in P.$$

A liveness property is true for every finite behavior. In contrast to safety properties, a liveness property P can never be refuted by observing only a finite prefix of some run in P, hence, in the words of LAMPORT, P states that "something good eventually happens".

For a liveness property P, it is obvious from the definition that every property $Q \supseteq P$ is also a liveness property. In particular, arbitrary unions of liveness properties yield liveness properties. In general, however, even finite intersection of liveness properties are not liveness properties. For example, consider two sets P and Q where:

- $P = \{\sigma \in \Sigma^\omega : \exists i \geq 0 : \forall j \geq i : \sigma[j] = s\}$ and
- $Q = \{\sigma \in \Sigma^\omega : \forall i \geq 0 : \exists j \geq i : \sigma[j] \neq s\}$,

where $s \in \Sigma$ is some designated state. It is easy to verify that both P and Q are liveness properties (assuming that Σ contains at least two different states) whereas $P \cap Q = \emptyset$ which is certainly not a liveness property.

3.2.3 Specification

In general, a specification is simply a property. However, it is usually convenient to give some structure to specifications.

To describe specifications formally, we represent a program by a state machine (or transition system) M and a supplementary property L, with intent that M defines the safety property of the specification and L defines the liveness property.

Definition 3.5 *(transition system) A transition system is a triple* $M = (\Sigma, I, R)$ *where*

- Σ *is a set (of states),*
- $I \subseteq \Sigma$ *is the set of initial states and*
- $R \subseteq \Sigma \times \Sigma$ *is the transition relation.*

The property defined by a transition system M, also denoted by M, is defined by

$$M = \{s_0, s_1, \ldots \in \Sigma^\omega : s_0 \in I \text{ and } (s_i, s_{i+1}) \in R \text{ for all } i \in \mathbb{N}\}.$$

A specification is a pair $S = (M, L)$ where $M = (\Sigma, I, R)$ is a transition system and $L \subseteq \Sigma^\omega$ is the supplementary property of S. The property defined by a specification S, which also denoted by S, is defined as

$$S = M \cap L.$$

A state machine M is identified with the specification (M, Σ^ω), whose supplementary property is trivial.

It is sometimes preferable to give some additional structure to the supplementary property itself. In particular, L may (in part) be given by sets of fairness constraints on actions. A *fair transition system* is a pair $F = (M, (\mathcal{W}, \mathcal{S}, L))$ where $M = (\Sigma, I, R)$ is a transition system, $L \subseteq \Sigma^\omega$ is a supplementary property as above, and $\mathcal{W}$ and $\mathcal{S}$ are subsets of $\Sigma \times \Sigma$. The property $F \subseteq \Sigma^\omega$ defined by a fair transition system F is defined by $\sigma \in F$ iff

- $\sigma \in M$
- for all $\alpha \in \mathcal{W}$, there are infinitely many $i \in \mathbb{N}$ such that $\sigma[i] \notin dom(\alpha)$ or $(\sigma[i], \sigma[i+1]) \in \alpha$
- for all $\alpha \in \mathcal{S}$, there exists some $i \in \mathbb{N}$ such that $\sigma[j] \notin dom(\alpha)$ holds for every $j \geq i$ or there are infinitely many $i \in \mathbb{N}$ such that $(\sigma[i], \sigma[i+1]) \in \alpha$
- $\sigma \in L$.

3.3 TLA*

The previous section dealt with a very general definition of property as arbitrary subsets of infinite runs. We now present TLA*, the logic that we use to describe our methods for representing and reasoning about reactive systems.

3.3.1 Propositional TLA* (pTLA*)

3.3.1.1 Syntax

Alphabet

The alphabet of pTLA* consists of the alphabet of propositional logic $\mathcal{L}_p$ and additional symbols $\Box, [,]$ and $\circ$.

Formulas and preformulas

Definition 3.6 *Formulas and preformulas of pTLA* are inductively defined as follows.*

1. **false** *is a formula, as is every atomic proposition* $v \in \mathcal{V}$.
2. *If* F, G *are formulas then* $F \to G$ *and* $\Box F$ *are formulas.*
3. *If* A *is a preformula and* $v \in \mathcal{V}$ *then* $\Box[A]_v$ *is a formula.*
4. *If* F *is formula then* F *and* $\circ F$ *are preformulas.*
5. *If* A, B *are preformulas then* $A \to B$ *is a preformula.*

For a (pre-)formula A*, we denote by* $At(A)$ *the set of atomic propositions that occur in* A.

For writing formulas, we will use symbols such as F, G for formulas and A, B for preformulas. Notice that $\Box[...]_v$ is considered to be a separate operator for all $v \in \mathcal{V}$.

Abbreviations
We use the abbreviation of propositional logic $\mathcal{L}_p$ for both formulas and preformulas. We sometimes write v' instead of $\circ v$ when $v \in \mathcal{V}$ is an atomic proposition. For (possibly primed) atomic propositions we sometimes use the equality symbol to denote equivalence, and write, for example $w' = v$ instead of $w' \leftrightarrow v$. If $V = \{v_1, \ldots, v_n\} \subseteq \mathcal{V}$ is a finite set of atomic propositions, we write $V' = V$ for the preformula $v_1' = v_1 \wedge \ldots \wedge v_n' = v_n$ and $\Box[A]_V$ or $\Box[A]_{v_1,\ldots,v_n}$ to denote the formula $\Box[A]_{v_1} \wedge \ldots \wedge \Box[A]_{v_n}$.

In particular, $\Box[A]_\emptyset$ is equal to **true**. We write $\Diamond F$ for the formula $\neg\Box\neg F$ and $\Diamond\langle A\rangle_v$ for $\neg\Box[\neg A]_v$. Consequently, $\Diamond\langle A\rangle_{v_1,\ldots,v_n}$ denotes $\Diamond\langle A\rangle_{v_1} \vee \ldots \vee \Diamond\langle A\rangle_{v_n}$.

3.3.1.2 Semantics

A *state* is a boolean valuation as in propositional logic, i.e. $s : \mathcal{V} \to \{\mathsf{tt}, \mathsf{ff}\}$ of the atomic propositions. A *behavior* $\sigma = s_0 s_1 \ldots$ is an infinite sequence of states. Notice that since a behavior is a sequence, the conventional notations for a sequence (described in Section 2.3) also hold for it.

We now define what it means for a preformula or a formula to hold of a behavior σ, written $\sigma \not\approx A$ or $\sigma \models F$, respectively. Because every formula is also preformula, this also defines the semantics of formulas.

Definition 3.7 *(semantics of (pre-)formulas) The semantics of preformulas is given by the relation $\not\approx$, which is inductively defined as follows:*

$\sigma \not\approx$ **false**.
$\sigma \approx v$ *iff* $s_0(v) = \mathsf{tt}$ *(for $v \in \mathcal{V}$).*
$\sigma \approx A \to B$ *iff* $\sigma \approx A$ *implies* $\sigma \approx B$.
$\sigma \approx \Box F$ *iff* $\sigma[i..] \approx F$ *holds for all* $i \in \mathbb{N}$.
$\sigma \approx \Box[A]_v$ *iff for all* $i \in \mathbb{N}, s_i(v) = s_{i+1}(v)$ *or* $\sigma[i..] \approx A$.
$\sigma \approx \circ F$ *iff* $\sigma[1..] \approx F$.

For a formula F, we usually write $\sigma \models F$ instead of $\sigma \approx F$.

A behavior σ such that $\sigma \not\approx A$ holds is called a *model* of A. We say that a formula F is *valid over a behavior* σ iff $\sigma[n..] \models F$ holds for all $n \in \mathbb{N}$. Finally, F is *valid* (written $\models F$) iff it is valid over all behaviors.

3.3.1.3 Stuttering invariance

pTLA* is invariant under stuttering. In this section we describe what it means for a logic to be invariant under stuttering.

Definition 3.8 *Let $V \subseteq \mathcal{V}$.*

1. *Two states s, t are called V-similar, written $s =_V t$, iff $s(v) = t(v)$ for all $v \in V$. Two behaviors σ, τ are called V-similar iff $s_i =_V t_i$ holds for all i.*

2. *V-stuttering equivalence, written $\simeq_V$, is the smallest equivalence relation on behaviors that identifies $\rho \circ \langle s \rangle \circ \sigma$ and $\rho \circ \langle tu \rangle \circ \sigma$, for any finite sequence of states ρ, infinite sequence of states σ, and pairwise V-similar states s, t, u.*

3. *Stuttering equivalence (written $\simeq$) is $\mathcal{V}$-stuttering equivalence.*

It follows that $\sigma \simeq_V \tau$ implies $\sigma \simeq_W \tau$ whenever $W \subseteq V$. In particular, stuttering equivalence is the finest relation among all $\simeq_V$. Note also that two states s, t are $\mathcal{V}$-similar iff they are equal.

A logic is invariant under stuttering when none of its formulas can distinguish between two stuttering-equivalent behaviors.

Given two behaviors $\sigma = s_0 s_1 \ldots$ and $\tau = t_0 t_1 \ldots$ such that $\sigma \simeq_V \tau$ holds then

1. $t_0 =_V s_0$.

2. For every $n \in \mathbb{N}$ there is some $m \in \mathbb{N}$ such that both $\sigma[n..] \simeq_V \tau[m..]$ and $\sigma[n+1..] \simeq_V \tau[m+1..]$.

Replacing maximal finite repetitions of V-similar states by a single occurrence of, say, the first of these states, it follows that for any behavior σ there exists a V-stuttering equivalent behavior $\natural_V \sigma = s_0 s_1 \ldots$ that contains successive V-similar states only if it ends in V-stuttering, that is, for all $i \in \mathbb{N}, s_i =_V s_{i+1}$ holds only if $s_i =_V s_j$ for all $j \geq i$. We call any such behavior $\natural_V \sigma$ a *V-stuttering free variant of σ*. Two behaviors σ, τ are V-stuttering equivalent if and only if $\natural_V \sigma =_V \natural_V \tau$ holds, where $\natural_V \sigma$ and $\natural_V \tau$ denote arbitrary V-stuttering free variants of σ and τ.

The following theorem can be used to prove that every pTLA* formula is invariant under stuttering.

Theorem 3.9 *(stuttering invariance) Assume that A is a preformula, F is a formula and that σ, τ are behaviors.*

1. *If $\sigma \simeq_{At(A)} \tau$ and $\sigma[1..] \simeq_{At(A)} \tau[1..]$ then $\sigma \not\approx A$ iff $\tau \not\approx A$.*
2. *If $\sigma \simeq_{At(F)} \tau$ then $\sigma \models F$ iff $\tau \models F$.*

The proof of Theorem 3.9 can be found in [83].

3.3.2 Quantified TLA* (qpTLA*)

3.3.2.1 Syntax

We now formally define the extension of TLA* by quantification over proposition variables (qpTLA*).

Definition 3.10 *(syntax of qpTLA*) The (pre-)formulas of qpTLA* are given inductively as in Definition 3.6, except by adding the following clause:*

6. *If F is a formula and $x \in \mathcal{V}$ is an atomic proposition then $\exists\, x : F$ is a formula.*

Occurrence of variables

We use standard conventions regarding free and bound propositions in formulas and preformulas. In particular, we define the set $At(G)$, for $F \equiv \exists\, v : G$, to be $At(G) \setminus \{v\}$, since v becomes bound by the quantifier.

Substitution
If F, G are formulas and x is proposition then $F[G/x]$ denoted the formula obtained from F by replacing every free occurrence of x in F by the formula G. If any propositions free in G would become bound by this replacement, we silently assume that bound propositions in F are suitably renamed to avoid such capture of free propositions.

Quantification
Informally, the formula $\mathbf{\exists}\, x \,:\, F$ is true of behavior σ if F is true for some behavior τ that differs from σ by the values assigned to x. In other words, quantification is over an ω-sequence of values rather than over single values. We also introduce quantification over values. We thus define (for both formulas and preformulas)

$$\exists\, x \,:\, F \equiv F[\mathbf{true}/x] \vee F[\mathbf{false}/x]$$

using different symbols to distinguish (rigid) quantification over values from (flexible) quantification. In either case, universal quantification is defined in the standard way as the dual of existential quantification:

$$\mathbf{\forall}\, x \,:\, F \equiv \neg \mathbf{\exists}\, x \,:\, \neg F \text{ and } \forall\, x \,:\, F \equiv \neg \exists\, x \,:\, \neg F.$$

3.3.2.2 Semantics

Definition 3.11 *For $x \in \mathcal{V}$ we define the relations $=_x$ and $\approx_x$ on behaviors as follows:*

1. *Two behaviors $\sigma = s_0 s_1 \ldots$ and $\tau = t_0 t_1 \ldots$ are equal up to x, written $\sigma =_x \tau$ if $s_i(v) = t_i(v)$ for all $i \geq 0$ and $v \in \mathcal{V}$, except possibly x.*

2. *The relation $\approx_x$, called similarity up to x, is defined as $\approx_x = (\simeq \circ =_x \circ \simeq)$, where $\simeq$ is stuttering equivalence and $\circ$ denotes relational composition.*

The semantics of quantification is now defined in terms of $\approx_x$ instead of $=_x$ in order to ensure invariance under stuttering.

Definition 3.12 *(semantics of qpTLA*) The semantics of qpTLA* is obtained by adding the following clause to Definition 3.7*

$\sigma \models \mathbf{\exists}\, x \,:\, F$ *iff* $\tau \models F$ *holds for some* $\tau \approx_x \sigma$.

Definition 3.12 ensures that formulas of qpTLA* are invariant under stuttering equivalence.

Theorem 3.13 *For any qpTLA* formula F and behaviors σ and τ where $\sigma \simeq_{At(F)} \tau$,*

$$\sigma \models F \text{ iff } \tau \models F.$$

3.3.3 First order TLA*

The extension of propositional TLA* to a full first-order temporal logic is essentially straightforward. In order to simplify the notation, in the first order TLA*, primed flexible variables included in the base syntax of first order TLA*. Otherwise, (rigid) quantification would be necessary to refer to the value of a variable at the successor state.

3.3.3.1 Syntax

Assume given a language $\mathcal{L}(\mathcal{F}, \mathcal{P})$ of first-order logic that defines sets $\mathcal{F}$ and $\mathcal{P}$ of function and predicate symbols, each equipped with an arity $n \geq 0$. Assume further given a set $\mathcal{X} = \mathcal{X}_r \cup \mathcal{X}_f$ of variables, partitioned into disjoint sets $\mathcal{X}_r$ and $\mathcal{X}_f$ of *rigid* and *flexible* variables. (We assume that $\mathcal{X}$ does not contain symbols of the form x'.) The terms and formulas of TLA* are defined much as in classical first order logic; we also define *transition terms* that may contain primed flexible variables and are used in the definition of preformulas.

Terms
The terms of TLA* are inductively defined as follows.

1. Every variable $x \in \mathcal{X}$ is a term.
2. If $t_1, \ldots, t_n$ are terms and f is an n-ary function symbol then $f(t_1, \ldots, t_n)$ is a term. For $n = 0$ we usually write f instead of $f()$.

Transition terms
The transition terms of TLA* are defined similarly:

1. Every variable $x \in \mathcal{X}$ is a transition term. For a flexible variable $v \in \mathcal{X}_f$, also v' is a transition term.
2. If $t_1, \ldots, t_n$ are transition terms and f is an n-ary function symbol then $f(t_1, \ldots, t_n)$ is a transition term.

For a term t, we denote by t' the transition term obtained from t by replacing every flexible variable v in t by v'.

Formulas and preformulas
Formulas and preformulas of TLA* are inductively defined as follows.

1. **false** is a formula.
2. If $t_1, \ldots, t_n$ are terms and P is an n-ary predicate symbol then $P(t_1, \ldots, t_n)$ is a formula. If t_1 and t_2 are terms then $t_1 = t_2$ is a formula.
3. If F, G are formulas then $F \rightarrow G$ and $\Box F$ are formulas.
4. If A is a preformula and t is a term then $\Box[A]_t$ is a formula.
5. If F is a formula and $x \in \mathcal{X}_r$ is a rigid variable then $\exists x : F$ is a formula.
6. If F is a formula and $x \in \mathcal{X}_f$ is a flexible variable then $\mathbf{\exists}\, x : F$ is a formula.
7. If F is a formula then F and $\circ F$ are preformulas.
8. If $t_1, \ldots, t_n$ are transition terms and P is an n-ary predicate symbol then $P(t_1, \ldots, t_n)$ is a preformula. If t_1 and t_2 are transition terms then $t_1 = t_2$ is a preformula.
9. If A, B are preformulas then $A \rightarrow B$ is a preformula.
10. If A is a preformula and $x \in \mathcal{X}_r$ is a rigid variable then $\exists x : A$ is a preformula.

Occurrence of variables and substitutions
Free and bound occurrences of variable, and the sets $FFV(F)$ and $FRV(F)$ of free (flexible or rigid) variables that occur in (pre-)formula F are defined in the standard way. Note that $\mathbf{\exists}$ binds unprimed as well as primed occurrences of the quantified flexible variable.

We use notation analogous to that introduced for (quantified) propositional TLA*. For example, $\Box[F]_{t_1,\ldots,t_n}$ denotes the formula

$$\Box[F]_{t_1} \wedge \ldots \wedge \Box[F]_{t_n}.$$

We now write $\Box[A]_F$, for a formula F and a preformula A, to denote the formula $\Box[A]_{FFV(F)}$. Substitution is now defined at the level of terms. The substitution $F[t/x]$ replaces t' for x' in action subformulas of F.

State and action formulas

State formulas are those formulas that do not contain any temporal operators. Similarly *action formulas* (actions for short) are those preformulas that are built from (possible primed) variables, function and predicate symbols (including $=$), and first-order connectives, but that do not contain any temporal connectives. We sometimes write P', for a state formula P, to denote the action formula obtained from P by replacing every free flexible variable v in P by its primed counterpart v'. If A is an action we write

$$\textsc{enabled}\ A \equiv \exists\, v'_1, \ldots, v'_n\ :\ A$$

to denote the state formula obtained from A existentially quantifying over all free primed flexible variables $v'_1, \ldots, v'_n$ in A[1]. Furthermore, if t is a term (or finite set of terms), we write

$$\mathrm{WF}_t(A) \equiv \Diamond\Box\textsc{enabled}\ \langle A\rangle_t \rightarrow \Box\Diamond\langle A\rangle_t$$

$$\mathrm{SF}_t(A) \equiv \Box\Diamond\textsc{enabled}\ \langle A\rangle_t \rightarrow \Box\Diamond\langle A\rangle_t$$

3.3.3.2 Semantics

As in classical first order logic, the semantics of TLA* is defined relative to a first-order interpretation I that provides a universe $|I|$ and, for all function and predicate symbols $f \in \mathcal{F}$ and $P \in \mathcal{P}$, interpretations f^I and P^I, which are functions and relations of suitable arity. A *state* is now a valuation $s : \mathcal{X}_f \rightarrow |I|$ of the flexible variables. A *behavior* is an infinite sequence $\sigma = s_0 s_1 \ldots$ of states.

For a set $X \subseteq \mathcal{X}_f$ of flexible variables, the relations $=_X$ and $\simeq_X$ are defined in definitions 3.8. Similarly, for $x \in \mathcal{X}_f$ the relations $=_X$ and $\approx$ are defined as in definition 3.11.

Term semantics

Given an interpretation I and a valuation of the rigid variables $\alpha : \mathcal{X}_r \rightarrow |I|$, the semantics $s[\![t]\!]^{I,\alpha}u$ of a transition term t at states s and u is inductively defined as follows:

$$\begin{aligned}
s[\![x]\!]^{I,\alpha}u &= \alpha(x) && \text{for } x \in \mathcal{X}_r\\
s[\![v]\!]^{I,\alpha}u &= s(v) && \text{for } v \in \mathcal{X}_f\\
s[\![v']\!]^{I,\alpha}u &= u(v) && \text{for } v \in \mathcal{X}_f\\
s[\![f(t_1,\ldots,t_n)]\!]^{I,\alpha}u &= f^I(s[\![t_1]\!]^{I,\alpha}u, \ldots, s[\![t_n]\!]^{I,\alpha}u).
\end{aligned}$$

[1]Note that we use rigid quantification over $v'_1, \ldots, n'_n$. To be truly formal, we should replace all primed occurrences of flexible variables by fresh rigid variables or define the syntax as in [2]

Because every term is a transition term, the above definition also defines the semantics of terms. Clearly, the second state u is irrelevant for the semantics of terms; we will therefore often write $s[\![t]\!]^{I,\alpha}$ when t is a term.

Formula semantics
The semantics of (pre-)formulas, relative to an interpretation I, a valuation $\alpha : \mathcal{X}_r \to |I|$, and a behavior $\sigma = s_0 s_1 \ldots$ is inductively defined as follows:

$$
\begin{array}{rcl}
I, \sigma, \alpha \not\approx \mathbf{false} & & \\
I, \sigma, \alpha \approx P(t_1, \ldots, t_n) & \text{iff} & (s_0[\![t_1]\!]^{I,\alpha} s_1, \ldots, s_0[\![t_n]\!]^{I,\alpha} s_1) \in P^I. \\
I, \sigma, \alpha \approx t_1 = t_2 & \text{iff} & s_0[\![t_1]\!]^{I,\alpha} s_1 = s_0[\![t_2]\!]^{I,\alpha} s_1 \\
I, \sigma, \alpha \approx F \to G & \text{iff} & I, \sigma, \alpha \approx F \text{ implies } I, \sigma, \alpha \approx G \\
I, \sigma, \alpha \approx \Box F & \text{iff} & I, \sigma[i..], \alpha \approx F \text{ holds for all } i \in \mathbb{N} \\
I, \sigma, \alpha \approx \Box[A]_t & \text{iff} & \text{for all } i \in \mathbb{N}, s_i[\![t]\!]^{I,\sigma} = s_{i+1}[\![t]\!]^{I,\alpha} \text{ or} \\
 & & \sigma[i..] \approx A \\
I, \sigma, \alpha \approx \exists\, x : F & \text{iff} & I, \sigma, \beta \approx F \text{ for some valuation } \beta \text{ such that} \\
 & & \beta(y) = \alpha(y) \text{ for all } y \neq x \\
I, \sigma, \alpha \approx \mathbf{\exists}\, v : F & \text{iff} & I, \tau, \alpha \approx F \text{ for some behavior } \tau \simeq_v \sigma \\
I, \sigma, \alpha \approx \circ F & \text{iff} & I, \sigma[1..], \alpha \approx F
\end{array}
$$

If F is a formula, we usually write $I, \sigma, \alpha \models F$ instead of $I, \sigma, \alpha \approx F$.

The above definition ensures that formulas of first-order TLA* are again invariant under stuttering equivalence.

Theorem 3.14 *Assume that I is a first-order interpretation, σ and τ are behaviors, and α is a valuation of the rigid variables. For any formula F of first-order TLA*, if $\sigma \simeq_{FFV(F)} \tau$ then $I, \sigma, \alpha \models F$ iff $I, \tau, \alpha \models F$.*

3.3.4 Specifications

We now describe the structure of specifications that will be used in this book for representing reactive systems.

The general form of specification is given by a formula of the form:

$$\exists x : Init \wedge \Box[Next]_v \wedge L \tag{3.1}$$

where

- x is a list of internal variable,
- *Init* is a state predicate that describes the initial states,

- *Next* is an action characterizes the system's next-state relation,
- v is a state function, and
- L is a formula stating the liveness conditions expected from the system.

This formula essentially describes a state machine, augmented by liveness condition, as described in Subsection 3.2.3, that generates the allowed behaviors of the system under specification.

Usually, v will be the tuple of all variables appearing free in $Init, Next$ and L (including the variables of x). It follows from the definitions that a behavior satisfies Formula (3.1) iff there is some way of choosing values for x such that

1. *Init* is true in the initial state,
2. every step is either an *Next* step or leaves all the variables in v unchanged and
3. the entire behavior satisfies L.

Note that Formula 3.1 is the general TLA specification formula proposed by Lamport [70]. In fact, using TLA* we can write specifications, which are more complex than this formula. However, we prefer to restrict to this form due to its simplicity and it has been shown that this form is expressive enough to represent the classes of systems we will consider, namely discrete, real-time and parameterized systems.

3.3.5 Machine closed

A finite sequence of states is called a finite behavior. For any formula F and finite behavior τ, we say that τ satisfies F iff τ can be extended to an infinite behavior that satisfies F. It can be shown that, for any TLA* formula F, there is a TLA* formula $\mathcal{C}(F)$, called the *closure* of F, such that a behavior σ satisfies $\mathcal{C}(F)$ iff every prefix of σ satisfies F. Formula $\mathcal{C}(F)$ is the strongest safety property such that $\models F \rightarrow \mathcal{C}(F)$.

If M is a safety property and L is a supplementary property, then the pair (M, L) is *machine closed* iff every finite behavior satisfying M can be extended to an infinite behavior in $M \cap L$. The lack of machine closure can be a source of incompleteness for proof methods. As illustration, given a safety property F and we want to prove that $M \cap L$ satisfies F. Most methods for proving safety properties use only safety properties as hypotheses, so we can

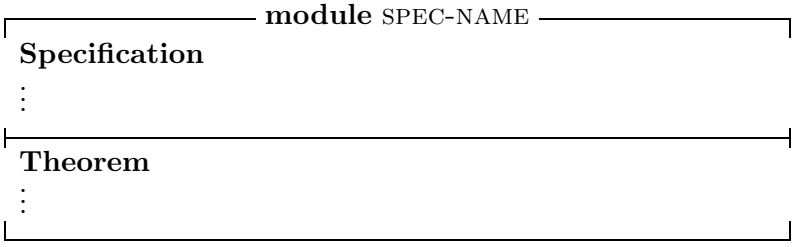

Figure 3.1: A module.

prove $M \cap L \rightarrow F$ only by proving $M \rightarrow F$. If M is not machine closed, then $M \cap L \rightarrow F$ could hold even though $M \rightarrow F$ does not, and these methods will be unable to prove that system with specification M satisfies F.

We recall the canonical form of our specification is of the form of $\exists\, x : Init \wedge \Box[Next]_v \wedge L$ as stated in Formula (3.1). Thus, in order to have a machine closed specification, we expect L to constrain infinite behaviors. This means, formally, that the closure of Formula (3.1) should be the formula $Init \wedge \Box[Next]_v$.

3.4 Writing specifications

When writing a specification, we sometimes use the TLA$^+$'s writing style [73], shown in Figure 3.1 where

- **module** is a keyword,
- SPEC-NAME represents the name of our specification and
- **Specification** is a keyword starting a list of formulas describing the behavior of the system.
- **Theorem** is a keyword starting a list of theorems representing the properties we want to prove.

We also use the notation that of a list of expressions bulleted by $\wedge$ denotes their conjunction and a list of expressions bulleted by $\vee$ denotes their disjunction. For a tuple v, UNCHANGED v represents $v' = v$, $v[i]$ represents the i^{th} component of v, $[v$ EXCEPT $!\, i = e]$ asserts that the i^{th} component is equal to e and $v' = [v$ EXCEPT $!\, i = e]$ asserts that except the i^{th} component that is equal to e, the rest of components of v' is equal to the ones of v.

For formulas A, B and C we sometimes write

$$\textbf{if } A \textbf{ then } B \qquad \text{and} \qquad \textbf{if } A \textbf{ then } B \textbf{ else } C$$

for formula $A \rightarrow B$ and $A \rightarrow B \wedge \neg A \rightarrow C$, respectively.

As illustration we take the so called "increment problem" [71]. We specify a system that starts with x and y both equal to 0 and repeatedly increments x and y by 1. A step increments either x or y (but not both). The variables are incremented in arbitrary order, but each is incremented infinitely often. This system might be represented in a conventional programming language as in Figure 3.2.

```
initial x = 0, y = 0;
cobegin
  loop forever                    loop forever
    x := x + 1          ||          y := y + 1
  end loop                        end loop
coend
```

Figure 3.2: Increment problem: Pseudocode representation.

The specification for the increment problem is given in Figure 3.3. Notice that we assume x and y are natural numbers and we want to prove that the system preserves this property.

3.5 Remarks

The notion of safety and liveness properties have been first introduced by Lamport [68]. Informally, a safety property expresses that "something (bad) will *not* happen" during a system execution. A liveness property expresses that eventually "something (good) *must* happen" during an execution. The distinction of safety and liveness properties was motivated by the different techniques for proving those properties. For example, Owicki & Lamport [90] proposed the technique of proof lattices for liveness properties. Later, in [3] Lamport made his informal characterization of safety property more precise. A property is called a safety property iff each execution violating the property has a finite prefix violating that property, and vice versa, if a finite prefix of an execution violates the property then the execution itself violates the property. This corresponds to the intuition that the "bad thing" (i.e. violating the property) can be detected in a finite initial part of the

module INCREMENT

Specification

$$
\begin{array}{rl}
X \equiv & \wedge\ x' = x + 1 \\
& \wedge\ \text{UNCHANGED}\ \langle y \rangle \\
Y \equiv & \wedge\ y' = y + 1 \\
& \wedge\ \text{UNCHANGED}\ \langle x \rangle \\
Inc \equiv & \wedge\ x = 0 \wedge y = 0 \\
& \wedge\ \Box[X \vee Y]_{\langle x,y \rangle} \\
& \wedge\ \text{WF}_{\langle x,y \rangle}(X) \wedge \text{WF}_{\langle x,y \rangle}(Y)
\end{array}
$$

Theorem

1. $Inc \rightarrow \Box(x, y \in \mathbb{N})$

Figure 3.3: Module INCREMENT.

execution and the occurrence of the "bad thing" in a prefix of an execution is irremediable.

ALPERN & SCHNEIDER [4] were the first to give a formal definition of both safety and liveness properties. In contrast to LAMPORT, they represented a finite prefix of an execution as the set of all possible continuations from that point on, which leads to a slighty more general notion of safety properties. A property is a liveness property, iff it contains at least one continuation for every finite prefix. This corresponds to the intuition that the "good thing" (i.e. satisfying the property) can still happen after any finite execution.

Another classification of properties is also given by MANNA & PNUELI in [76, 81]. They introduced a hierarchy of properties which agrees with the classification given here on the safety properties. More issues on the classification of properties can be found, for example, in [5, 6, 100, 97] and [60].

The generalized Temporal Logics of Actions (TLA*) is proposed by MERZ [83]. It is a variant of linear-time temporal logic, inspired by LAMPORT's Temporal Logic of Actions [69, 71]. In the other direction, TLA can be viewed as the sub-logic of TLA* obtained by restricting preformulas to be actions. As in TLA, in TLA*, systems as well as their properties are represented by formula. The difference from TLA is that it is based on a symmetrical, mutually recursive definition of (stuttering sensitive) preformulas and (stuttering invariant) formulas rather than a layer of formulas on top of a layer of non-temporal action formulas. We choose this logic because of its complete axiomatization which will be very beneficial for deductive verification.

More issues on machine closed can be consulted, for example, in [1].

Chapter 4

Automata on infinite words

4.1 Overview

Automata theory plays an important role in computer science. Various kinds of automata are used, for example, for compilation, natural language analysis, complexity theory and hardware design. Automata theory also fits well in the domain of modeling and verification of systems.

A finite automaton is a mathematical model of a device that has a constant amount of memory, independent of the size of its input [32]. Modeling systems using transition systems and using automata to specify properties of the systems, i.e. using the same kind of representations to describe both systems and their properties, brings a lot benefits. One can then perform automatic verification by exploring graph algorithms.

Automata that operates on infinite objects such as ω-words, trees or graphs have been important tools for the analysis of temporal and related logics since the pioneering work by Büchi [24], Muller [86], McNaughton [84], Rabin [96], and Vardi&Wolper [109].

In this chapter, we will consider a class of finite automata over infinite words, called Muller automata [86]. First, we give the formal definition of Muller automata. An algorithm for translating pTLA* formulas into Muller automata will be given in Section 4.3, which is the main contribution of this chapter. After presenting timed automata, which is the extension of ω-automata with a finite set of real-valued *clocks* in Section 4.4, we conclude this chapter by giving a brief discussion and some related work.

4.2 Muller automata

Definition 4.1 ***(Muller automaton)*** *A Muller automaton $\mathcal{M}$ over an alphabet $\Sigma_\mathcal{M}$ is a tuple $(Q,\ Q_0, \Delta, \mathcal{F})$ such that*

- *Q is a finite set of locations,*[1]
- *$Q_0 \subseteq Q$ is a finite set of initial locations,*
- *$\Delta \subseteq Q \times \Sigma_\mathcal{M} \times Q$ is a transition relation and*
- *$\mathcal{F} \subseteq 2^Q$ is a set of sets of locations.*

A run of $\mathcal{M}$ over an ω-word $w = x_0x_1 \ldots \in \Sigma^\omega_\mathcal{M}$ is an infinite sequence $\pi = q_0q_1 \ldots$ of locations $q_i \in Q$ such that $q_0 \in Q_0$ and $(q_i, x_i, q_{i+1}) \in \Delta$ holds for all $i \in \mathbb{N}$.

Let $\inf(\pi)$ denote the set of locations which appear infinitely often in π. A run $\pi = q_0q_1 \ldots$ is accepting if $\inf(\pi) \in \mathcal{F}$.

A word $w = x_0x_1 \ldots$ is called accepted if there exists an accepting run of $\mathcal{M}$ over w.

The language $\mathcal{L}(\mathcal{M}) \subseteq \Sigma^\omega_\mathcal{M}$ is the set of accepted words.

In a graph representing an automaton we mark initial locations with an incoming arrow that is not connected to any other node. An example of Muller automaton is given in Figure 4.1. The acceptance set is given by $\{\{q_1\}, \{q_0, q_1\}\}$. This automaton accepts all words over the alphabet $\Sigma_\mathcal{M} = \{a, b\}$ that contain b infinitely often.

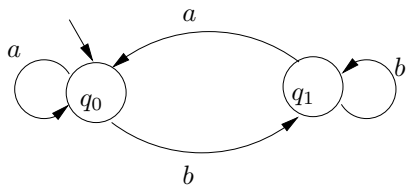

Figure 4.1: An example of Muller automaton.

Given two Muller automata $\mathcal{M}_1$ and $\mathcal{M}_2$ over $\Sigma_\mathcal{M}$, one can build an automaton that accepts $\mathcal{L}(\mathcal{M}_1) \cap \mathcal{L}(\mathcal{M}_2)$ using Construction 4.2.

[1] We use the term locations rather than conventional states to avoid confusion with the states of transition systems and temporal logic.

Construction 4.2 *(product automaton) Let* $\mathcal{M}_1 = (Q_1, Q_1^0, \Delta_1, \mathcal{F}_1)$ *and* $\mathcal{M}_2 = (Q_2, Q_2^0, \Delta_2, \mathcal{F}_2)$ *be two Muller automata over* $\Sigma_{\mathcal{M}}$*. The product of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *is an automaton* $\mathcal{M}^p = (Q^p, Q_0^p, \Delta^p, \mathcal{F}^p)$ *over* $\Sigma_{\mathcal{M}}$ *such that*

1. $Q^p = Q^1 \times Q^2$*;*
2. $Q_0^p = Q_0^1 \times Q_0^2$*;*
3. $((q_1, q_2), x, (q_1', q_2')) \in \Delta^p$ *iff* $(q_1, x, q_1') \in \Delta_1$ *and* $(q_2, x, q_2') \in \Delta_2$*; and*
4. $\mathcal{F}^p = \mathcal{F}_1^p \cap \mathcal{F}_2^p$ *where* $\mathcal{F}_1^p = \mathcal{F}_1 \times Q_2$ *and* $\mathcal{F}_2^p = Q_1 \times \mathcal{F}_2$*.*

4.3 From pTLA* to Muller-automata

We now consider the following problem: given a pTLA* (pre-)formula A, construct a Muller automaton that accepts exactly the behaviors satisfying A.

The central part of the automaton construction algorithm is a tableau-like procedure related to the ones described in [51, 32, 93]. This procedure builds a graph, which will define the locations and transitions of the automaton.

For a given (pre-)formula A we first construct the *formula graph* of A, which is a rooted graph whose every node contains a set of sub-(pre-)formulas of A representing all sub-(pre-)formulas of A that can hold together on a position in the model. Two nodes n_1 and n_2 are connected with a directed edge (n_1, n_2) if the (pre-)formulas in n_2 can hold at a state following one that satisfies the (pre-)formulas in n_1.

Then the automaton can be constructed by using this graph and defining the suitable accepting conditions.

4.3.1 Graph construction

The main idea behind the construction is that any (pre-)formula can be decomposed to give a collection of sets containing (pre-)formulas that either apply to the current time alone or are (pre-)formulas that holds at the next time.

PNP

In the following, we use a structure called PNP (positive-negative-pair) for representing (pre-)formulas.

Definition 4.3 (PNP) *A* PNP *is a pair* $\mathrm{P} = (\mathcal{F}^+, \mathcal{F}^-)$ *of two finite sets* $\mathcal{F}^+$ *and* $\mathcal{F}^-$ *of (pre-)formulas. The formula* $\widehat{\mathrm{P}}$ *will be the abbreviation of*

$$\bigwedge_{A \in \mathcal{F}^+} A \wedge \bigwedge_{B \in \mathcal{F}^-} \neg B.$$

We denote by $\mathcal{F}(\mathrm{P})$ *the set containing every (pre-)formula in* $\mathcal{F}^+$ *and the negation form of every (pre-)formula in* $\mathcal{F}^-$.
A PNP *is called inconsistent if* **false** $\in \mathcal{F}^+$ *or there exists some (pre-)formula* A *such that* $A \in \mathcal{F}^+$ *and* $A \in \mathcal{F}^-$, *i.e.* $\mathcal{F}^+ \cap \mathcal{F}^- \neq \emptyset$.

As convention a PNP $(\emptyset, \emptyset)$ represents the formula **true**. We also define some operations over PNPs.

Definition 4.4 *Given two* PNP*s* $\mathrm{P}_1 = (\mathcal{F}_1^+, \mathcal{F}_1^-)$ *and* $\mathrm{P}_2 = (\mathcal{F}_2^+, \mathcal{F}_2^-)$, *the operation* $\cup, \cap$ *and* $\setminus$ *over* P_1 *and* P_2 *are defined as follows:*

1. $\mathrm{P}_1 \cup \mathrm{P}_2 = (\mathcal{F}_1^+ \cup \mathcal{F}_2^+, \mathcal{F}_1^- \cup \mathcal{F}_2^-)$.
2. $\mathrm{P}_1 \cap \mathrm{P}_2 = (\mathcal{F}_1^+ \cap \mathcal{F}_2^+, \mathcal{F}_1^- \cap \mathcal{F}_2^-)$.
3. $\mathrm{P}_1 \setminus \mathrm{P}_2 = (\mathcal{F}_1^+ \setminus \mathcal{F}_2^+, \mathcal{F}_1^- \setminus \mathcal{F}_2^-)$.

Node
The data structure we use for representing nodes contains sufficient information for the graph construction algorithm to be able to operate in a DFS order.

Definition 4.5 ***(node)*** *A node* $\mathcal{N}$ *is given by a tuple* $(Init, Loc, Exp, Old, Next)$ *where:*

- *Init is a boolean variable indicating whether* $\mathcal{N}$ *is an initial location or not,*
- *Loc is a boolean variable indicating whether* $\mathcal{N}$ *is a location or not,*
- *Exp is a* PNP *representing a set of (pre-)formulas that must hold at the current node and have not yet been processed,*
- *Old is a* PNP *representing a set of (pre-)formulas that must hold at the current node and have been already processed and*

- *Next is a* PNP *representing a set of (pre-)formulas that are promised to hold in the successors of the current node.*

For a node $\mathcal{N}$ we use the notation $\mathcal{N}_{Init}, \mathcal{N}_{Loc}, \mathcal{N}_{Exp}, \mathcal{N}_{Old}$ and $\mathcal{N}_{Next}$ for referring its components.

A node $\mathcal{N}$ is called a *candidate location*, or *location* for short, if both $pos(\mathcal{N}_{Exp})$ and $neg(\mathcal{N}_{Exp})$ are empty and called a *root* if $pos(\mathcal{N}_{Old})$, $neg(\mathcal{N}_{Old})$, $pos(\mathcal{N}_{Next})$ and $neg(\mathcal{N}_{Next})$ are empty.

$\mathcal{N}_{Init}$	$\mathcal{N}_{Loc}$
$pos(\mathcal{N}_{Exp})$	$neg(\mathcal{N}_{Exp})$
$pos(\mathcal{N}_{Old})$	$neg(\mathcal{N}_{Old})$
$pos(\mathcal{N}_{Next})$	$neg(\mathcal{N}_{Next})$

Figure 4.2: Graphical representation of a node.

Expansion rule

Depending on its form, a (pre-)formula will be expanded during the expansion step. Every row of the Table 4.1 represents an expansion rule, which is based on the definition of semantics of pTLA* (pre-)formulas and can be read as follows:

$$\widehat{\mathrm{P}} \leftrightarrow \bigvee_{i=1..k} \widehat{New_i(\mathrm{P})} \wedge \circ \widehat{Next_i(\mathrm{P})}$$

where $k \in 1..3$ depends on the form of the (pre-)formula. For example, if P is a PNP of the form $(\{\Box A\}, \emptyset)$ then the expansion rule is:

$$\begin{aligned} \widehat{\mathrm{P}} \quad &\leftrightarrow \widehat{New_1(\mathrm{P})} \wedge \circ \widehat{Next_1(\mathrm{P})} \\ &\leftrightarrow A \wedge \circ \Box A. \end{aligned}$$

Expansion step

Given a PNP P, the formula graph for $\widehat{\mathrm{P}}$ can be constructed by using algorithm FORMULA-GRAPH in Figure 4.3. The input parameters of this procedure are two nodes $\mathcal{N}^s$ and $\mathcal{N}^c$, a set of nodes V and a set of edges E.

Initially, $\mathcal{N}^s$ is a dummy node which is set to $\langle$FALSE, FALSE, $(\emptyset, \emptyset)$, $(\emptyset, \emptyset)$, $(\emptyset, \emptyset)\rangle$. During the execution $\mathcal{N}^s$ will represent a source node of the edges

P	$New_1(\mathrm{P})$	$Next_1(\mathrm{P})$	$New_2(\mathrm{P})$	$Next_2(\mathrm{P})$	$New_3(\mathrm{P})$	$Next_3(\mathrm{P})$
$(\{\circ F\}, \emptyset)$	$-$	$(\{F\}, \emptyset)$	$-$	$-$	$-$	$-$
$(\{A \to B\}, \emptyset)$	$(\emptyset, \{A\})$	$(\emptyset, \emptyset)$	$(\{B\}, \emptyset)$	$(\emptyset, \emptyset)$	$-$	$-$
$(\{\Box A\}, \emptyset)$	$(\{A\}, \emptyset)$	$(\{\Box A\}, \emptyset)$	$-$	$-$	$-$	$-$
$(\{\Box[A]_v\}, \emptyset)$	$(\{v\}, \emptyset)$	$(\{v, \Box[A]_v\}, \emptyset)$	$(\emptyset, \{v\})$	$(\{\Box[A]_v\}, \{v\})$	$(\{A\}, \emptyset)$	$(\{\Box[A]_v\}, \emptyset)$
$(\emptyset, \{\circ F\})$	$-$	$(\emptyset, \{F\})$	$-$	$-$	$-$	$-$
$(\emptyset, \{A \to B\})$	$(\{A\}, \{B\})$	$(\emptyset, \emptyset)$	$-$	$-$	$-$	$-$
$(\emptyset, \{\Box A\})$	$(\emptyset, \{A\})$	$(\emptyset, \emptyset)$	$(\emptyset, \emptyset)$	$(\emptyset, \{\Box A\})$	$-$	$-$
$(\emptyset, \{\Box[A]_v\})$	$(\{v\}, \{A\})$	$(\emptyset, \{v\})$	$(\emptyset, \{v, A\})$	$(\{v\}, \emptyset)$	$(\emptyset, \emptyset)$	$(\emptyset, \{\Box[A]_v\})$

Table 4.1: Expansion table.

produced by the algorithm. $\mathcal{N}^c$ is initially set to the node whose *Exp* component is P. This node becomes the initial node or the root of the formula graph. The pair (V, E) represents the formula graph.

Using a dummy node D, an initial node R, and V, E, which are initially empty, as parameters, the algorithm calls the procedure EXPAND in Figure 4.4.

Every time the procedure EXPAND is called, it checks the consistency of the *Exp* component of $\mathcal{N}^c$ (line 1). If $\mathcal{N}^c_{Exp}$ is not inconsistent then it continues to check whether V already contains a node $\mathcal{N}^n$ whose *Exp*, *Old* and *Next* components are the same to the ones of $\mathcal{N}^c$ (lines 2-3). If it is so then a new edge that connects the source node $\mathcal{N}^s$ to $\mathcal{N}^n$ is added to E (line 4). Otherwise, the current node $\mathcal{N}^c$ is stored to V (line 6) and if $\mathcal{N}^c_{Init}$ is equal to FALSE then a new edge that connects the source node $\mathcal{N}^s$ to the current node $\mathcal{N}^c$ is inserted into E (line 7). A new edge is not created if $\mathcal{N}^c_{Init}$ = TRUE, since $\mathcal{N}^s$ is the dummy node D as long as $\mathcal{N}^c$ = TRUE. The procedure then checks whether the current node is a location or not (line 8). If yes then the $\mathcal{N}^c_{Loc}$ becomes TRUE (line 9) and furthermore, since there are no more unprocessed obligations left in $\mathcal{N}^c_{Exp}$, a new node $\mathcal{N}^n$ is created (line 10) and the procedure EXPAND is called with $\mathcal{N}^c, \mathcal{N}^n$, V and E as parameters (line 11).

In case of $\mathcal{N}^c$ is not a location, $\mathcal{N}^c$ is expanded to produce one (lines 13-28), two (lines 29-36) or three (lines 37-46) successor node(s). In this step the algorithm uses the information in Table 4.1. The first part of Table 4.1 (rows 1-4) is related to the (pre-)formulas that can be represented as positive component of a PNP P, whereas the second part (rows 5-8) is related to the ones that can be represented as negative components of a PNP P.

Figure 4.6.a illustrates the graph construction for formula $\Box p$. Every node is marked with a number representing the ordering of its generation. The nodes, which are locations, are drawn with bold lines. Nodes with some incoming edges drawn as arrows crossed by a double line are roots. The underlined (pre-)formulas are (pre-)formulas on which expansion rules have been applied.

4.3.2 Automaton definition

Now the graph constructed by the algorithm FORMULA-GRAPH can be used to define a Muller automaton that accepts the infinite words that satisfy the (pre-)formula $\widehat{\mathrm{P}}$. The set of locations Q of the automaton contains all the nodes $\mathcal{N}$ in V such that $\mathcal{N}_{Loc}$ = TRUE. The initial locations Q_0 are those locations $\mathcal{N}$ in Q such that $\mathcal{N}_{Init}$ = TRUE.

Algorithm FORMULA-GRAPH

Input: a (pre-)formula $\widehat{R_{Exp}}$; Output: a formula graph (V, E).

1 $D = \langle \text{TRUE}, \text{FALSE}, (\emptyset, \emptyset), (\emptyset, \emptyset), (\emptyset, \emptyset) \rangle$
2 $R = \langle \text{TRUE}, \text{FALSE}, (R^+_{Exp}, R^-_{Exp}), (\emptyset, \emptyset), (\emptyset, \emptyset) \rangle$
3 $V = \emptyset$
4 $E = \emptyset$
5 EXPAND(D, R, V, E)

Figure 4.3: Formula graph generation algorithm.

As the input alphabet for the automaton, $\Sigma_{\mathcal{M}}$, we take the set containing sets of atomic propositions that occur in $\widehat{\text{P}}$, i.e. $\Sigma_{\mathcal{M}} = 2^{At(\widehat{\text{P}})}$. We will later regard every element x of $\Sigma_{\mathcal{M}}$ as a state, i.e. Boolean valuation over atomic propositions in $At(\widehat{\text{P}})$, $s : At(\widehat{\text{P}}) \to \{\mathsf{tt}, \mathsf{ff}\}$, such that for every atomic proposition v in x, $s(v) = \mathsf{tt}$ and $s(v) = \mathsf{ff}$, otherwise.

Definition 4.6 ***(positive-negative occurrences of atomic proposition)*** *Let q be a location in Q. We denote by $Pos(q)$ and $Neg(q)$ the set containing all atomic propositions in q^+_{Old} and q^-_{Old}, i.e., $Pos(q)$ and $Neg(q)$ is the positive and negative occurrences of the propositions in q_{Old}, respectively.*

The transitions of the automaton are defined in a way such that for two locations $\mathcal{N}$ and $\mathcal{N}'$ in Q and $x \in \Sigma_{\mathcal{M}}$, $(\mathcal{N}, x, \mathcal{N}')$ is in Δ iff there exists a path in E from $\mathcal{N}$ to $\mathcal{N}'$ without traversing any other locations in Q and x satisfies the atomic propositions contained in $\mathcal{N}_{Old}$, i.e. $At(\widehat{\text{P}}) \supseteq x \supseteq Pos(\mathcal{N})$ and $x \cap Neg(\mathcal{N}) = \emptyset$.

The important step in the automaton definition is the definition of accepting conditions. We observe that not every maximal path $\pi = q^0 q^1 \ldots$ in the constructed graph determines models of the (pre-)formula. For instance, the construction allows some nodes to contain a (pre-)formula of the form $\neg\Box A$ while none of the successor nodes contains $\neg A$.

Definition 4.7 ***(promising* PNP*)*** *A* PNP P *is called a promising* PNP *if its form appears in the left column of Table 4.2.*

Algorithm EXPAND

Input : a node $\mathcal{N}^s$, $\mathcal{N}^c$, a set of nodes V and a set of edges E .

if $\mathcal{N}^c_{Exp}$ is not inconsistent **then**
  if there exists some node $\mathcal{N}^n \in V$ such that $\mathcal{N}^n_{Exp} = \mathcal{N}^c_{Exp}, \mathcal{N}^n_{Old} = \mathcal{N}^c_{Old}$
    and $\mathcal{N}^n_{Next} = \mathcal{N}^c_{Next}$ **then**
    $E = E \cup \{(\mathcal{N}^s, \mathcal{N}^n)\}$
  else
    $V = V \cup \{\mathcal{N}^c\}$
    if $\mathcal{N}_{Init} =$ FALSE **then** $E = E \cup \{(\mathcal{N}^s, \mathcal{N}^c)\}$ **endif**
    if $\mathcal{N}^c$ is a location **then**
      $\mathcal{N}^c_{Loc} =$ TRUE
      $\mathcal{N}^1 = \langle$FALSE, FALSE, $\mathcal{N}^c_{Next}, \emptyset, \emptyset\rangle$
      EXPAND($\mathcal{N}^c, \mathcal{N}^1, V, E$)
    else
      if $\mathcal{N}^c_{Exp}$ contains a PNP P of the form $(\{v\}, \emptyset)$ or $(\emptyset, \{v\})$ for some $v \in \mathcal{V}$
      then
        $\mathcal{N}^1 = \langle \mathcal{N}^c_{Init}, \text{FALSE}, \mathcal{N}^c_{Exp} \setminus \text{P}, \mathcal{N}^c_{Old} \cup \text{P}, \mathcal{N}^c_{Next}\rangle$
        EXPAND($\mathcal{N}^c, \mathcal{N}^1, V, E$)
      endif
        if$\mathcal{N}^c_{Exp}$ contains a PNP P of the form $(\{\circ G\}, \emptyset)$ or $(\emptyset, \{\circ G\})$ for
        some formula G **then**
        $\mathcal{N}^1 = \langle \mathcal{N}^c_{Init}, \text{FALSE}, \mathcal{N}^c_{Exp} \setminus \text{P}, \mathcal{N}^c_{Old} \cup \text{P}, \mathcal{N}^c_{Next} \cup Next_1(\text{P})\rangle$
        EXPAND($\mathcal{N}^c, \mathcal{N}^1, V, E$)
      endif
      if $\mathcal{N}^c_{Exp}$ contains a PNP P of the form $(\{\Box A\}, \emptyset)$ or $(\emptyset, \{A \rightarrow B\})$ for
        some (pre-)formulas A and B and some atomic proposition $v \in \mathcal{V}$
      then
        $\mathcal{N}^1 = \langle \mathcal{N}^c_{Init}, \text{FALSE}, \mathcal{N}^c_{Exp} \setminus \text{P} \cup New_1(\text{P}) \setminus \mathcal{N}^c_{Old}, \mathcal{N}^c_{Old} \cup \text{P}, \mathcal{N}^c_{Next} \cup Next_1(\text{P})\rangle$
        EXPAND($\mathcal{N}^c, \mathcal{N}^1, V, E$)
      endif
      if $\mathcal{N}^c_{Exp}$ contains a PNP P of the form $(\{A \rightarrow B\}, \emptyset)$ or $(\emptyset, \{\Box A\})$ for
        some (pre-)formulas A and B and some atomic proposition $v \in \mathcal{V}$
      then
        $\mathcal{N}^1 = \langle \mathcal{N}^c_{Init}, \text{FALSE}, \mathcal{N}^c_{Exp} \setminus \text{P} \cup New_1(\text{P}) \setminus \mathcal{N}^c_{Old}, \mathcal{N}^c_{Old} \cup \text{P}, \mathcal{N}^c_{Next} \cup Next_1(\text{P})\rangle$
        EXPAND($\mathcal{N}^c, \mathcal{N}^1, V, E$)
        $\mathcal{N}^2 = \langle \mathcal{N}^c_{Init}, \text{FALSE}, \mathcal{N}^c_{Exp} \setminus \text{P} \cup New_2(\text{P}) \setminus \mathcal{N}^c_{Old}, \mathcal{N}^c_{Old} \cup \text{P}, \mathcal{N}^c_{Next} \cup Next_2(\text{P})\rangle$
        EXPAND($\mathcal{N}^c, \mathcal{N}^2, V, E$)
      endif
      if $\mathcal{N}^c_{Exp}$ contains a PNP P of the form $(\{\Box[A]_v\}, \emptyset)$ or $(\emptyset, \{\Box[A]_v\})$
        for some (pre-)formula A and atomic proposition $v \in \mathcal{V}$ **then**

Figure 4.4: EXPAND algorithm.

Algorithm EXPAND

```
39          N^1 = ⟨N^c_Init, FALSE, N^c_Exp \ P ∪ New_1(P) \ N^c_Old, N^c_Old ∪ P, N^c_Next ∪ Next_1(P)⟩
40          EXPAND(N^c, N^1, V, E)
41          N^2 = ⟨N^c_Init, FALSE, N^c_Exp \ P ∪ New_2(P) \ N^c_Old, N^c_Old ∪ P, N^c_Next ∪ Next_2(P)⟩
42          EXPAND(N^c, N^2, V, E)
43          N^3 = ⟨N^c_Init, FALSE, N^c_Exp \ P ∪ New_3(P) \ N^c_Old, N^c_Old ∪ P, N^c_Next ∪ Next_3(P)⟩
44          EXPAND(N^c, N^3, V, E)
45        endif
46      endif
47    endif
48  endif
```

Figure 4.5: EXPAND algorithm (continued).

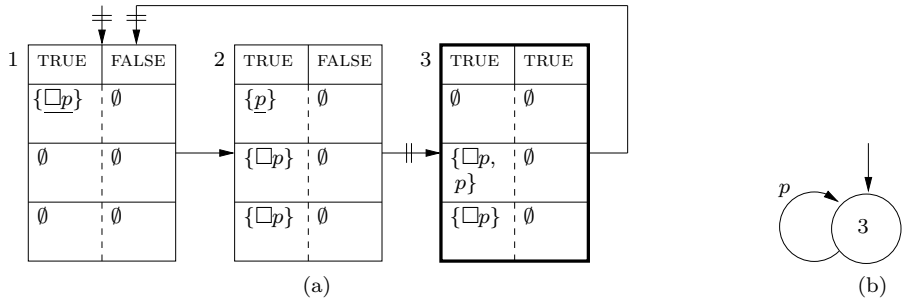

Figure 4.6: (a) Formula graph and (b) Muller automaton for $\Box p$.

P	$Old_1(\mathrm{P})$	$Next_1(\mathrm{P})$	$Old_2(\mathrm{P})$	$Next_2(\mathrm{P})$
$(\emptyset, \{\Box A\})$	$(\emptyset, \emptyset)$	$(\emptyset, \{A\})$	–	–
$(\emptyset, \{\Box[A]_v\})$	$(\{v\}, \{A\})$	$(\emptyset, \{v\})$	$(\emptyset, \{A, v\})$	$(\{v\}, \emptyset)$

Table 4.2: Promising and fulfilling PNPs.

Definition 4.8 ***(fulfilling node and formula)*** *Let* P *be a promising* PNP*. Then a node* $\mathcal{N}$ *is called a fulfilling node of* P *if one of the following conditions hold*

- P *is of the form* $(\emptyset, \{\Box A\})$ *and* $\mathcal{F}(Old_1(\mathrm{P})) \subseteq \mathcal{F}(\mathcal{N}_{Old})$ *and* $\mathcal{F}(Next_1(\mathrm{P})) \subseteq \mathcal{F}(\mathcal{N}_{Next})$*.*
- P *is of the form* $(\emptyset, \{\Box[A]_v\})$ *and either* $\mathcal{F}(Old_1(\mathrm{P})) \subseteq \mathcal{F}(\mathcal{N}_{Old})$ *and* $\mathcal{F}(Next_1(\mathrm{P})) \subseteq \mathcal{F}(\mathcal{N}_{Next})$ *or* $\mathcal{F}(Old_2(\mathrm{P})) \subseteq \mathcal{F}(\mathcal{N}_{Old})$ *and* $\mathcal{F}(Next_2(\mathrm{P})) \subseteq \mathcal{F}(\mathcal{N}_{Next})$*.*

For a promising PNP P*, we also call the formula(s)* $\widehat{Old_1(\mathrm{P})} \wedge \circ\widehat{Next_1(\mathrm{P})}$ *(and* $\widehat{Old_2(\mathrm{P})} \wedge \circ\widehat{Next_2(\mathrm{P})}$*) the fulfilling formula(s) of* P*.*

We now define the so-called *accepting* SCSs based on two definitions above.

Definition 4.9 ***(accepting*** SCS***)*** *Let* $\mathcal{S}$ *be the set containing all the* SCS*s of* Q *and* $\mathcal{R}$ *be the set containing all promising* PNP*s contained in Old component of every node in* Q*. We define* $\mathcal{F}$ *to be the set containing all* SCS*s in* $\mathcal{S}$ *such that for every* $F \in \mathcal{F}$ *and for every* $\mathrm{P} \in \mathcal{R}$*, if there exists some location* $\mathcal{N}$ *in* F *such that if* $\mathrm{P} \in \mathcal{N}_{Old}$ *then there exists some location* $\mathcal{N}'$ *which is a fulfilling node of* P*.*

Construction 4.10 *(graph to Muller automaton) Let* P *be a* PNP *and* V *and* E *be a set of nodes and edges returned by the Algorithm* FORMULA-GRAPH*. The Muller automaton* $\mathcal{M}$ *over* $2^{At(\widehat{\mathrm{P}})}$ *which accepts exactly the models of* P *is given by* $(Q, Q_0, \Delta, \mathcal{F})$ *where*

- $Q \subseteq V$ *such that for every* q *in* V*,* q *is in* Q *iff* $q_{Loc} =$ TRUE*,*
- $Q_0 \subseteq Q$ *such that for every* q *in* Q*,* q *is in* Q_0 *iff* $q_{Init} =$ TRUE*,*

- $\Delta \subseteq Q \times 2^{At(\widehat{\mathrm{P}})} \times Q$ *such that* $(q^1, x, q^2) \in \Delta$ *if there exists a path from* $q^i \ldots q^j$ *such that* $q^i = q^1, q^j = q^2$ *and for every* $k, i < k < j, q^k_{Loc} =$ FALSE *and* $At(\widehat{\mathrm{P}}) \supseteq x \supseteq Pos(q^1)$ *and* $x \cap Neg(q^1) = \emptyset$.
- $\mathcal{F} \subseteq 2^Q$ *is as defined in Definition 4.9.*

We say that the construction succeeds if $\mathcal{F}$ is not empty whenever $\mathcal{R}$ is not empty. Otherwise, we say that the construction fails, which implies that (pre-)formula P is unsatisfiable.

Figure 4.6.b illustrates the constructed automaton for formula $\Box p$. To relate the automaton with the formula graph in Figure 4.6.a, each location is labeled with q and the ordering number in the formula graph. For labeling the transitions, we consider the set of atomic propositions contained in the *Old* component of the source location. In the case of the *Old* component of the source location does not contain any atomic proposition, we label the transition with **true**. For example, the loop is labeled with atomic proposition p since the *Old* component of the q_3 contains p. The acceptance set of this automaton is $\{\{q_3\}\}$.

4.3.3 Proof of correctness

In this section, we will prove the correctness of the algorithms explained in the previous section.

Theorem 4.11 *Let* $\mathcal{M}$ *be the constructed automaton for (pre-)formula A. Then a behavior* $\sigma = s_0 s_1 \ldots$ *is a model of A iff there exists an accepting run* $\pi = q^0 q^1 \ldots$ *such that* $\sigma[i..] \models \widehat{q^i_{Old}} \wedge \circ\widehat{q^i_{Next}}$ *holds for every* $i \in \mathbb{N}$.

The two directions are proven in Lemma 4.17 and Lemma 4.18. Note that in the following proofs, we only consider the (pre-)formulas that can be represented as the positive components of PNPs. The other (pre-)formulas can be proven similarly.

Lemma 4.12 *(node-splitting)*

1. *When a node* $\mathcal{N}$ *is split during the construction in lines 29-36 in* EXPAND *algorithm into two nodes* $\mathcal{N}_1$ *and* $\mathcal{N}_2$, *the following condition holds:*

$$\widehat{\mathcal{N}_{Old}} \wedge \widehat{\mathcal{N}_{Exp}} \wedge \circ\widehat{\mathcal{N}_{Next}} \longleftrightarrow \begin{array}{l} \widehat{\mathcal{N}^1_{Old}} \wedge \widehat{\mathcal{N}^1_{Exp}} \wedge \circ\widehat{\mathcal{N}^1_{Next}} \vee \\ \widehat{\mathcal{N}^2_{Old}} \wedge \widehat{\mathcal{N}^2_{Exp}} \wedge \circ\widehat{\mathcal{N}^2_{Next}} \end{array}$$

2. *When a node $\mathcal{N}$ is split during the construction in lines 37-46 in* EXPAND *algorithm into three nodes $\mathcal{N}_1, \mathcal{N}_2$ and $\mathcal{N}_3$, the following holds:*

$$\widehat{\mathcal{N}_{Old}} \wedge \widehat{\mathcal{N}_{Exp}} \wedge \circ\widehat{\mathcal{N}_{Next}} \longleftrightarrow \begin{array}{l} \widehat{\mathcal{N}^1_{Old}} \wedge \widehat{\mathcal{N}^1_{Exp}} \wedge \circ\widehat{\mathcal{N}^1_{Next}} \vee \\ \widehat{\mathcal{N}^2_{Old}} \wedge \widehat{\mathcal{N}^2_{Exp}} \wedge \circ\widehat{\mathcal{N}^2_{Next}} \vee \\ \widehat{\mathcal{N}^3_{Old}} \wedge \widehat{\mathcal{N}^3_{Exp}} \wedge \circ\widehat{\mathcal{N}^3_{Next}} \end{array}$$

3. *When a node $\mathcal{N}$ is updated to become a new node $\mathcal{N}^1$, as in lines 13-28 in* EXPAND *algorithm the following holds:*

$$\widehat{\mathcal{N}_{Old}} \wedge \widehat{\mathcal{N}_{Exp}} \wedge \circ\widehat{\mathcal{N}_{Next}} \longleftrightarrow \widehat{\mathcal{N}^1_{Old}} \wedge \widehat{\mathcal{N}^1_{Exp}} \wedge \circ\widehat{\mathcal{N}^1_{Next}}$$

Proof. Directly from EXPAND algorithm and the definition of the semantics of pTLA*. ■

Let $\mathcal{R}_t$ denote the set of all the roots of the constructed formula graph for A. Then for every root $\mathcal{N}$ in $\mathcal{R}_t$ one of the following conditions holds:

1. $\mathcal{N}$ is the initial node, R, on line 2 in Algorithm FORMULA-GRAPH in Figure 4.3.

2. $\mathcal{N}$ is obtained at line 10 in Algorithm EXPAND in Figure 4.4 from some node $\mathcal{N}'$ whose construction is finish. Thus, we have $\mathcal{N}_{Exp} = \mathcal{N}'_{Next}$.

From the algorithm EXPAND, every root $\mathcal{N}$ is propagated to produce some nodes $\mathcal{N}^1, \ldots, \mathcal{N}^k$ such that $\mathcal{N}^i_{Exp} = (\emptyset, \emptyset)$. We call such nodes the *same-time descendant nodes* of $\mathcal{N}$. Moreover, since for every $i \in 1..k$, $\mathcal{N}^i_{Loc} =$ TRUE, these nodes become the locations of the constructed automaton.

Lemma 4.13 *Let $\mathcal{N}^0$ be a root and $\mathcal{N}^1, \ldots, \mathcal{N}^k$ be its same-time descendant nodes. Then*

$$\widehat{\mathcal{N}^0_{Exp}} \longleftrightarrow \bigvee_{i \in 1..k} \widehat{\mathcal{N}^i_{Old}} \wedge \circ\widehat{\mathcal{N}^i_{Next}}.$$

Proof. Suppose $\widehat{\mathcal{N}^0_{Exp}}$ represents a formula consisting only one single formula F, i.e. $\mathcal{F}(\mathcal{N}^0_{Exp}) = F$.

Case 1: $\mathcal{N}^0_{Exp}$ is of the form $(\{F\}, \emptyset)$.

$F = v$ for $v \in \mathcal{V}$.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce one new node $\mathcal{N}^1$ such that $\mathcal{N}^1_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^1_{Old} = (\{v\}, \emptyset)$ and $\mathcal{N}^1_{Next} = (\emptyset, \emptyset)$. $\mathcal{N}^1$ is the only same-time descendant node of $\mathcal{N}^0$.

$F = \circ v$ for $v \in \mathcal{V}$.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce one new node $\mathcal{N}^1$ such that $\mathcal{N}^1_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^1_{Old} = (\{\circ v\}, \emptyset)$ and $\mathcal{N}^1_{Next} = (\{v\}, \emptyset)$. $\mathcal{N}^1$ is the only same-time descendant node of $\mathcal{N}^0$.

$F = A \rightarrow B$ for A and B some (pre-)formulas.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce two new nodes $\mathcal{N}^1$ such that $\mathcal{N}^1_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^1_{Old} = (\emptyset, \{A\})$ and $\mathcal{N}^1_{Next} = (\emptyset, \emptyset)$; and $\mathcal{N}^2$ such that $\mathcal{N}^2_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^2_{Old} = (\{B\}, \emptyset)$ and $\mathcal{N}^2_{Next} = (\emptyset, \emptyset)$. $\mathcal{N}^1$ and $\mathcal{N}^2$ are the same-time descendant nodes of $\mathcal{N}^0$.

$F = \Box A$ for some (pre-)formula A.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce one new node $\mathcal{N}^1$ such that $\mathcal{N}^1_{Exp} = (\{A\}, \emptyset)$, $\mathcal{N}^1_{Old} = (\{\Box A\}, \emptyset)$ and $\mathcal{N}^1_{Next} = (\{\Box A\}, \emptyset)$. Applying Lemma 4.12 once again, $\mathcal{N}^1$ will be propagated to produce one new node $\mathcal{N}^2$ such that $\mathcal{N}^2_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^2_{Old} = (\{\Box A, A\}, \emptyset)$ and $\mathcal{N}^2_{Next} = (\{\Box A\}, \emptyset)$. $\mathcal{N}^2$ is the only same-time descendant node of $\mathcal{N}^0$.

$F = \Box[A]_v$ for some (pre-)formula A and some atomic proposition $v \in \mathcal{V}$.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce three new nodes $\mathcal{N}^1, \mathcal{N}^2$ and $\mathcal{N}^3$ such that

- $\mathcal{N}^1_{Exp} = (\{v\}, \emptyset)$, $\mathcal{N}^1_{Old} = (\{\Box[A]_v\}, \emptyset)$, $\mathcal{N}^1_{Next} = (\{v, \Box[A]_v\}, \emptyset)$,
- $\mathcal{N}^2_{Exp} = (\emptyset, \{v\})$, $\mathcal{N}^2_{Old} = (\{\Box[A]_v\}, \emptyset)$, $\mathcal{N}^2_{Next} = (\{\Box[A]_v\}, \{v\})$,
- $\mathcal{N}^3_{Exp} = (\{A\}, \emptyset)$, $\mathcal{N}^3_{Old} = (\{\Box[A]_v\}, \emptyset)$ and $\mathcal{N}^3_{Next} = (\{\Box[A]_v\}, \emptyset)$.

Furthermore, by Lemma 4.12, $\mathcal{N}^1$ will be propagated to produce one new node $\mathcal{N}^4$ such that $\mathcal{N}^4_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^4_{Old} = (\{\Box[A]_v, v\}, \emptyset)$ and $\mathcal{N}^4_{Next} = (\{v, \Box[A]_v\}, \emptyset)$; $\mathcal{N}^2$ will be propagated to produce one new node $\mathcal{N}^5$ such that $\mathcal{N}^5_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^5_{Old} = (\{\Box[A]_v\}, \{v\})$ and $\mathcal{N}^5_{Next} = (\{\Box[A]_v\}, \{v\})$; whereas $\mathcal{N}^3$ will be propagated to produce one new node $\mathcal{N}^6$ such that $\mathcal{N}^6_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^6_{Old} = (\{\Box[A]_v, A\}, \emptyset\})$ and $\mathcal{N}^6_{Next} = (\{\Box[A]_v\}, \emptyset\})$. $\mathcal{N}^4, \mathcal{N}^5$ and $\mathcal{N}^6$ are the same-time descendant nodes of $\mathcal{N}^0$.

Case 2: $\mathcal{N}^0_{Exp}$ is of the form $(\emptyset, \{F\})$.

$F = v$ for $v \in \mathcal{V}$.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce one new node $\mathcal{N}^1$ such that $\mathcal{N}^1_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^1_{Old} = (\emptyset, \{v\})$ and $\mathcal{N}^1_{Next} = (\emptyset, \emptyset)$. $\mathcal{N}^1$ is the only same-time descendant node of $\mathcal{N}^0$.

$F = \circ v$ for $v \in \mathcal{V}$.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce one new node $\mathcal{N}^1$ such that $\mathcal{N}^1_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^1_{Old} = (\emptyset, \{\circ v\})$ and $\mathcal{N}^1_{Next} = (\emptyset, \{v\})$. $\mathcal{N}^1$ is the only same-time descendant node of $\mathcal{N}^0$.

$F = A \rightarrow B$ for A and B some (pre-)formulas.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce one new node $\mathcal{N}^1$ such that $\mathcal{N}^1_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^1_{Old} = (\emptyset, \{A \rightarrow B\})$ and $\mathcal{N}^1_{Next} = (\{A\}, \{B\})$. $\mathcal{N}^1$ is the only same-time descendant node of $\mathcal{N}^0$.

$F = \Box A$ for some (pre-)formula A.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce two new nodes $\mathcal{N}^1$ and $\mathcal{N}^2$ such that $\mathcal{N}^1_{Exp} = (\emptyset, \{A\})$, $\mathcal{N}^1_{Old} = (\emptyset, \{\Box A\})$, $\mathcal{N}^1_{Next} = (\emptyset, \emptyset)$, $\mathcal{N}^2_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^2_{Old} = (\emptyset, \{\Box A\})$ and $\mathcal{N}^2_{Next} = (\emptyset, \{\Box A\})$. By Lemma 4.12, $\mathcal{N}^1$ will be propagated to produce one new node $\mathcal{N}^3$ such that $\mathcal{N}^3_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^3_{Old} = (\emptyset, \{\Box A, A\})$ and $\mathcal{N}^3_{Next} = (\emptyset, \emptyset)$. $\mathcal{N}^2$ and $\mathcal{N}^3$ are the same-time descendant nodes of $\mathcal{N}^0$.

$F = \Box[A]_v$ for some (pre-)formula A and some atomic proposition $v \in \mathcal{V}$.

By Lemma 4.12, $\mathcal{N}^0$ will be propagated to produce three new nodes $\mathcal{N}^1, \mathcal{N}^2$ and $\mathcal{N}^3$ such that

- $\mathcal{N}^1_{Exp} = (\{v\}, \{A\})$, $\mathcal{N}^1_{Old} = (\emptyset, \{\Box[A]_v\})$, $\mathcal{N}^1_{Next} = (\emptyset, \{v\})$,
- $\mathcal{N}^2_{Exp} = (\emptyset, \{v, A\})$, $\mathcal{N}^2_{Old} = (\emptyset, \{\Box[A]_v\})$, $\mathcal{N}^2_{Next} = (\{v\}, \emptyset)$
- $\mathcal{N}^3_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^3_{Old} = (\emptyset, \{\Box[A]_v\})$ and $\mathcal{N}^3_{Next} = (\emptyset, \{\Box[A]_v\})$.

Applying Lemma 4.12 for the second time, $\mathcal{N}^1$ will be propagated to produce one new node $\mathcal{N}^4$ such that $\mathcal{N}^4_{Exp} = (\emptyset, \{A\})$, $\mathcal{N}^4_{Old} = (\{v\}, \{\Box[A]_v\})$ and $\mathcal{N}^4_{Next} = (\emptyset, \{v\})$. $\mathcal{N}^2$ will be propagated to produce one new node $\mathcal{N}^5$ such that $\mathcal{N}^5_{Exp} = (\emptyset, \{A\})$, $\mathcal{N}^5_{Old} = (\emptyset, \{\Box[A]_v, v\})$ and $\mathcal{N}^5_{Next} = (\{v\}, \emptyset)$. Furthermore, Applying Lemma 4.12 for the third time, $\mathcal{N}^4$ and $\mathcal{N}^5$ will be propagated to produce $\mathcal{N}^6$ and $\mathcal{N}^7$ such that $\mathcal{N}^6_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^6_{Old} = (\{v\}, \{\Box[A]_v, A\})$, $\mathcal{N}^6_{Next} = (\emptyset, \{v\})$, $\mathcal{N}^7_{Exp} = (\emptyset, \emptyset)$, $\mathcal{N}^7_{Old} = (\emptyset, \{\Box[A]_v, v, A\})$ and $\mathcal{N}^7_{Next} = (\{v\}, \emptyset)$.

$\mathcal{N}^3, \mathcal{N}^6$ and $\mathcal{N}^7$ are the same-time descendant nodes of $\mathcal{N}^0$.

For all those cases we have $\widehat{\mathcal{N}^0_{Exp}} \longleftrightarrow \bigvee_{i \in 1..k} \widehat{\mathcal{N}^i_{Old}} \wedge \circ\widehat{\mathcal{N}^i_{Next}}$ where k is the number of the same-time descendant nodes of $\mathcal{N}^0$.

Assume we have a procedure $getSTDN1(\mathcal{N})$ that returns the set containing all same-time descendant nodes of some node $\mathcal{N}$ where $\widehat{\mathcal{N}_{Exp}}$ represents a formula consisting of one single formula.

Algorithm STDN

Input : a node $\mathcal{N}$, Output: a set of nodes S_0.

```
1   S_0 = ∅
1   for every F ∈ F(N_Exp) do
2     S = getSTDN1(F)
3     S_temp = S_0
4     S_0 = ∅
5     for every N^1 ∈ S_temp do
6       for every N^2 ∈ S do
7         S_0 = S_0 ∪ ⟨N_Init, TRUE, N^1_Exp ∪ N^2_Exp, N^1_Old ∪ N^2_Old, N^1_Next ∪ N^2_Next⟩
8       endfor
9     endfor
10  endfor
11  return S_0
```

Figure 4.7: Procedure STDN.

We consider the case where $\widehat{\mathcal{N}^0_{Exp}}$ represents a complex formula consisting more than one single formula, i.e. $|\mathcal{F}(\mathcal{N}^0_{Exp})| > 1$. In order to compute the same-time descendant nodes of $\mathcal{N}^0$ we can use procedure STDN in Figure 4.7. It can be shown that for this case we also have $\widehat{\mathcal{N}^0_{Exp}} \longleftrightarrow \bigvee_{i \in 1..k} \widehat{\mathcal{N}^i_{Old}} \wedge \circ \widehat{\mathcal{N}^i_{Next}}$ where k is the number of the same-time descendant nodes of $\mathcal{N}^0$. ■

Lemma 4.14 *Let $\mathcal{N}^0$ be a root whose Exp component contains some promising* PNP P *and $\sigma = s_0 s_1 \ldots$ be a behavior such that $\sigma \models\!\!\approx \widehat{\mathcal{N}^0_{Exp}}$. Let* P_1 *be one of the fulfilling* PNP*s of* P*. Then if $\sigma \models\!\!\approx \widehat{\mathrm{P}_1}$ then there exists some of the same-time descendant nodes of $\mathcal{N}^0$ whose Old component contains* P_1.

Proof. Let $\mathcal{N}^0$ be a root and P be some promising PNP contained by $\mathcal{N}^0_{Exp}$. We consider two cases:

1. $\mathrm{P} = (\{\Diamond A\}, \emptyset)$ for some (pre-)formula A.

   By Definition 4.8, the fulfilling formula of P is A, i.e. $\mathrm{P}_1 = (\{A\}, \emptyset)$. Since $\sigma \models\!\!\approx \widehat{\mathrm{P}}$ and $\sigma \models\!\!\approx \widehat{\mathrm{P}_1}$ by assumption, there must be some same-time descendant node of $\mathcal{N}^0$, $\mathcal{N}$, such that $(\{\Diamond A, A\}, \emptyset) \subseteq \mathcal{N}_{Old}$. In the proof of Lemma 4.13 we have shown that such node exists ($\mathcal{N}^3$).

2. $\mathrm{P} = (\{\Diamond \langle A \rangle_v\}, \emptyset)$ for some (pre-)formula A and some atomic proposition $v \in \mathcal{V}$.

By Definition 4.8, the fulfilling formula of P is $A \wedge v \wedge \circ\neg v$ or $A \wedge \neg v \wedge \circ v$, i.e the fulfilling node of P is a node $\mathcal{N}$ such that $(\{A, v\}, \emptyset) \subseteq \mathcal{N}_{Old}$ and $(\{v\}, \emptyset) \subseteq \mathcal{N}_{Next}$ or $(\{A\}, \{v\}) \subseteq \mathcal{N}_{Old}$ and $(\emptyset, \{v\}) \subseteq \mathcal{N}_{Next}$. In the proof of Lemma 4.13 we have shown that such nodes exist ($\mathcal{N}^6$ or $\mathcal{N}^7$). ■

Lemma 4.15 *Let $\sigma = s_i s_{i+1} \ldots$ be a behavior and q be a location of $\mathcal{M}$ such that $\sigma \models\!\!\approx \widehat{q_{Old}} \wedge \circ\widehat{q_{Next}}$. Then there exists a transition $(q, x, q') \in \Delta$ (for some $x \in \Sigma_{\mathcal{M}}$) such that $\sigma[i..] \models\!\!\approx \widehat{q'_{Old}} \wedge \circ\widehat{q'_{Next}}$. Moreover, if q_{Old} contains some promising* PNP P *but q_{Old} is not a fulfilling* PNP *of* P *and $\sigma[i+1..] \models\!\!\approx$ P$_1$ for* P$_1$ *is a fulfilling* PNP *of* P*, then in particular there exists a transition $(q, x, q') \in \Delta$ (for some $x \in \Sigma_{\mathcal{M}}$) such that q'_{Old} contains* P$_1$*. .*

Proof. Let $\sigma = s_i s_{i+1} \ldots$ be a behavior and q be a location of $\mathcal{M}$ such that $\sigma \models\!\!\approx \widehat{q_{Old}} \wedge \circ\widehat{q_{Next}}$. Since q is a location by assumption, then by the EXPAND algorithm a new root $\mathcal{N}$ is created (line 10) such that $\mathcal{N}_{Exp} = q_{Next}$. By Lemma 4.13, $\mathcal{N}$ will be expanded to produce some locations $\mathcal{N}^1, \ldots, \mathcal{N}^k$ such that

$$\widehat{\mathcal{N}_{Exp}} \longleftrightarrow \bigvee_{i \in 1..k} \widehat{\mathcal{N}^i_{Old}} \wedge \circ\widehat{\mathcal{N}^i_{Next}}.$$

Since $\mathcal{N}_{Exp} = q_{Next}$, this implies that

$$\widehat{q_{Next}} \longleftrightarrow \bigvee_{i \in 1..k} \widehat{\mathcal{N}^i_{Old}} \wedge \circ\widehat{\mathcal{N}^i_{Next}}.$$

By assumption $\sigma \models\!\!\approx \widehat{q_{Old}} \wedge \circ\widehat{q_{Next}}$. It follows that

$$\sigma[1..] \models\!\!\approx \bigvee_{i \in 1..k} \widehat{\mathcal{N}^i_{Old}} \wedge \circ\widehat{\mathcal{N}^i_{Next}}.$$

By EXPAND algorithm, for every $i \in 1..k$, there exists a path from q to $\mathcal{N}^i$. Thus, since every $\mathcal{N}^i$ is a location, by Construction 4.10, these edges will become the transitions of $\mathcal{M}$.

Now assume that q_{Old} contains a promising PNP P but q is not a fulfilling node of P and $\sigma[i+1..]$ satisfies one of the fulfilling formulas of P. Then Lemma 4.14 guarantees that there exists some of the same-time descendant nodes of $\mathcal{N}$, $\mathcal{N}^i$, such that $\mathcal{N}^i$ is a fulfilling node of P. ■

Lemma 4.16 *Let* P *be a* PNP *and $\mathcal{M}$ be the constructed automaton for $\widehat{\text{P}}$. Then*

$$\widehat{\text{P}} \longleftrightarrow \bigvee_{q \in Q_0} \widehat{q_{Old}} \wedge \circ\widehat{q_{Next}}.$$

Proof. By Lemma 4.13 with $\mathcal{N}^0$ is the initial node R. Since $R_{Exp} =$ P. ■

Lemma 4.17 *Let* P *be a* PNP*,* $\mathcal{M}$ *be the constructed automaton for* $\widehat{\mathrm{P}}$ *and* $\pi = q^0 q^1 \ldots$ *be an accepting run of* $\mathcal{M}$*. Then a behavior* $\sigma = s_0 s_1 \ldots$ *such that* $\sigma[i..] \models \widehat{q^i_{Old}} \wedge \circ\widehat{q^i_{Next}}$ *holds for every* $i \in \mathbb{N}$ *is a model of* $\widehat{\mathrm{P}}$*.*

Proof. Let P be a PNP, $\mathcal{M}$ be the constructed automaton for $\widehat{\mathrm{P}}$, $\pi = q^0 q^1 \ldots$ be an accepting run of $\mathcal{M}$ and $\sigma = s_0 s_1 \ldots$ be a behavior such that $\sigma[i..] \models \widehat{q^i_{Old}} \wedge \circ\widehat{q^i_{Next}}$ holds for every $i \in \mathbb{N}$.

Thus, $\mathcal{N}^0 = \langle \textsc{true}, \textsc{false}, \mathrm{P}, (\emptyset, \emptyset), (\emptyset, \emptyset) \rangle$. By EXPAND algorithm $\mathcal{N}^0$ will be expanded to produce some same-time descendant nodes, $\mathcal{N}^1, \ldots, \mathcal{N}^k$. Since $\mathcal{N}^0$ is the initial node, $\mathcal{N}^1, \ldots, \mathcal{N}^k$ will become the initial locations of $\mathcal{M}$. We consider the following cases:

- If **false** or $\neg$**false** is contained by $\mathcal{F}(\mathrm{P})$. Trivial.
- If $\mathcal{F}(\mathrm{P})$ contains a single formula F of the form v or $\neg v$ for $v \in \mathcal{V}$. Based on the proof of Lemma 4.13 it can be shown that for every initial location $q \in Q_0$, $F \in \mathcal{F}(q_{Old})$.
- If $\mathcal{F}(\mathrm{P})$ contains a formula F of the form $A \rightarrow B$ for A and B (pre-)formulas. Based on the proof of Lemma 4.13 it can be shown that for every initial location $q \in Q_0$, $\neg A \in \mathcal{F}(q_{Old})$ or $B \in \mathcal{F}(q_{Old})$.
- If $\mathcal{F}(\mathrm{P})$ contains a single formula F of the form $\neg(A \rightarrow B)$ for A and B (pre-)-formulas. Based on the proof of Lemma 4.13 it can be shown that for every initial location $q \in Q_0$, $\{\neg A, B\} \subseteq \mathcal{F}(q_{Old}$.
- If $\mathcal{F}(\mathrm{P})$ contains a single formula F of the form $\circ v$ or $\neg \circ v$ for $v \in \mathcal{V}$. Based on the proof of Lemma 4.13 it can be shown that for every initial location $q \in Q_0$, $F \in \mathcal{F}(q_{Next})$.
- If $\mathcal{F}(\mathrm{P})$ contains a single formula F of the form $\Box A$ for A some (pre-)formula.

  We will show that $A \in \mathcal{F}(q^i_{Old})$ holds for every $i \in \mathbb{N}$. It is shown in the proof of Lemma 4.13 that $A \in \mathcal{F}(q^0_{Old})$ and $\Box A \in \mathcal{F}(q^0_{Next})$.

  By EXPAND algorithm a new root $\mathcal{N}^n$ is created such that $\Box A \in \mathcal{F}(\mathcal{N}^n_{Exp})$. Repeatedly applying the similar argument for $q^1, q^2, \ldots$, we conclude that $A \in \mathcal{F}(q^i_{Old})$ and $A \in \mathcal{F}(q^i_{Next})$ holds for every $i \in \mathbb{N}$.
- If $\mathcal{F}(\mathrm{P})$ contains a single formula F of the form $\neg\Box A$ for A some (pre-)-formula.

  In this case we have to prove that there exists some $j \in \mathbb{N}$ such that $\neg\Box A \in \mathcal{F}(q^i_{Old})$ holds for every $i \leq j$, $\neg A \notin \mathcal{F}(q^i_{Old})$ holds for every $i < j$ and $\neg A \in \mathcal{F}(q^j_{Old})$.

  Referring to the proof of Lemma 4.13, either

1. $\{\neg\Box A, \neg A\} \subseteq \mathcal{F}(q^0_{Old})$ or
2. $\neg\Box A \in \mathcal{F}(q^0_{Old})$ and $\neg\Box A \in \mathcal{F}(q^0_{Next})$.

If the first case holds then the required condition is trivially satisfied. If the second case holds then by EXPAND algorithm, a new root $\mathcal{N}^n$ is created such that $\neg\Box A \in \mathcal{F}(\mathcal{N}^n_{Exp})$. Repeatedly applying the similar argument for $q^1, q^2, \ldots$, we conclude that for every $i \in \mathbb{N}$, either $\{\neg\Box A, \neg A\} \subseteq \mathcal{F}(q^i_{Old})$ or $\Box A \in \mathcal{F}(q^i_{Old})$ and $\neg\Box A \in \mathcal{F}(q^i_{Next})$. Since by assumption π is accepting, it is ensured that such j exists.

- If $\mathcal{F}(\text{P})$ contains a formula F of the form $\Box[A]_v$ for A some (pre-)formula and $v \in \mathcal{V}$.

  In this case it suffices to show that for every $i \in \mathbb{N}$, one of the following conditions hold:

  – $v \in \mathcal{F}(q^i_{Old})$ and $v \in \mathcal{F}(q^{i+1}_{Old})$,
  – $\neg v \in \mathcal{F}(q^i_{Old})$ and $\neg v \in \mathcal{F}(q^{i+1}_{Old})$ or
  – $A \in \mathcal{F}(q^i_{Old})$.

  Referring to the proof of Lemma 4.13, there are three possibilities for q^0, namely:

  1. $\{\Box[A]_v, v\} \subseteq \mathcal{F}(q^0_{Old})$ and $\{v, \Box[A]_v\} \subseteq \mathcal{F}(q^0_{Next})$,
  2. $\{\Box[A]_v, \neg v\} \subseteq \mathcal{F}(q^0_{Old})$ and $\{\Box[A]_v, \neg v\} \subseteq \mathcal{F}(q^0_{Next})$ or
  3. $\{\Box[A]_v, A\} \subseteq \mathcal{F}(q^0_{Old})$ and $\{\Box[A]_v\} \in \mathcal{F}(q^0_{Next})$.

  If condition 1 holds then by EXPAND algorithm a new root $\mathcal{N}^n$ is created such that $\{v, \neg\Box[A]_v\} \subseteq \mathcal{F}(\mathcal{N}^n)$. By EXPAND algorithm and Construction 4.10 either $\{\Box[A]_v, v\} \subseteq \mathcal{F}(q^1_{Old})$ and ($\{\Box[A]_v, v\} \subseteq \mathcal{F}(q^1_{Next})$ or $\{\Box[A]_v, v, A\} \subseteq \mathcal{F}(q^1_{Old})$ and $\Box[A]_v \in \mathcal{F}(q^1_{Next})$ holds. If the first case holds then by EXPAND algorithm a new root $\mathcal{N}^1$ is created such that $\{v, \neg\Box[A]_v\} \subseteq \mathcal{F}(\mathcal{N}^1_{Exp})$, otherwise $\Box[A]_v \in \mathcal{F}(\mathcal{N}^1_{Exp})$.

  If condition 2 holds then by EXPAND algorithm and Construction 4.10 either $\{\Box[A]_v, \neg v\} \subseteq \mathcal{F}(q^1_{Old})$ and $\{\Box[A]_v, \neg v\} \subseteq \mathcal{F}(q^1_{Next})$ or $\{\Box[A]_v, A, \neg v\} \subseteq \mathcal{F}(q^1_{Old})$ and $\Box[A]_v \in \mathcal{F}(q^1_{Next})$. If the first case holds then by EXPAND algorithm a new root $\mathcal{N}^1$ is created such that $\{\Box[A]_v, \neg v\} \subseteq \mathcal{N}^1_{Exp}$, otherwise $\Box[A]_v \in \mathcal{N}^1_{Exp}$.

  If condition 3 holds then by EXPAND algorithm a new root $\mathcal{N}^1$ is created such that $\Box[A]_v \in \mathcal{F}(\mathcal{N}^1_{Exp})$.

  Repeatedly applying the similar argument for $q^1, q^2, \ldots$, we conclude that the required condition is satisfied.

- If $\mathcal{F}(\text{P})$ contains a formula F of the form $\neg\Box[A]_v$ for A some (pre-)formula and $v \in \mathcal{V}$.

  In this case we have to prove that there exists some $j \in \mathbb{N}$ such that

  - $\neg\Box[A]_v \in \mathcal{F}(q^i_{Old})$ holds for every $i \leq j$,
  - neither $\{\neg A, \neg v\} \subseteq \mathcal{F}(q^i_{Old})$ and $v \in \mathcal{F}(q^{i+1}_{Old})$ nor $\{\neg A, v\} \subseteq \mathcal{F}(q^i_{Old})$ and $\neg v \in \mathcal{F}(q^{i+1}_{Old})$ holds for every $i < j$ and
  - $\neg A \in \mathcal{F}(q^j_{Old})$.

  Referring to the proof of Lemma 4.13, there are three possibilities for q^0, namely:

  1. $\{\neg\Box[A]_v, \neg A, \neg v\} \subseteq \mathcal{F}(q^0_{Old})$ and $v \in \mathcal{F}(q^0_{Next})$,
  2. $\{\neg\Box[A]_v, \neg A, v\} \subseteq \mathcal{F}(q^0_{Old})$ and $\neg v \in \mathcal{F}(q^0_{Next})$ or
  3. $\neg\Box[A]_v \in \mathcal{F}(q^0_{Old})$ and $\neg\Box[A]_v \in \mathcal{F}(q^0_{Old}) \in \mathcal{F}(q^0_{Next})$.

  If the first condition holds then by EXPAND algorithm a new root $\mathcal{N}^n$ is created such that $v \in \mathcal{F}(\mathcal{N}^n_{Exp})$. By EXPAND algorithm and Construction 4.10 $v \in \mathcal{F}(q^1_{Old})$. Thus the required condition is satisfied.

  If the second condition holds then by EXPAND algorithm a new root $\mathcal{N}^n$ is created such that $\neg \in \mathcal{F}(\mathcal{N}^n_{Exp})$. By EXPAND algorithm and Construction 4.10 $\neg v \in \mathcal{F}(q^1_{Old})$. Thus the required condition is satisfied.

  If the third condition holds then by EXPAND algorithm a new root $\mathcal{N}^n$ is created such that $\neg\Box[A]_v \in \mathcal{F}(\mathcal{N}^n_{Exp})$. Repeatedly applying the same argument for $q^1, q^2, \ldots$ we conclude that for every $i \in \mathbb{N}$ the condition $1, 2$ or 3 holds. Since by assumption π is accepting, it is ensured that such j exists.

■

Lemma 4.18 *Let* P *be a* PNP *and* $\mathcal{M}$ *be the constructed automaton for* $\widehat{\text{P}}$ *and* $\sigma = s_0 s_1 \ldots$ *be a model of* $\widehat{\text{P}}$. *Then there exists an accepting run of* $\mathcal{M}$ *such that* $\sigma[i..] \approx \widehat{q^i_{Old}} \wedge \circ\widehat{q^i_{Next}}$ *holds for every* $i \in \mathbb{N}$.

Proof. Let $\sigma = s_0 s_1 \ldots$ be a model of $\widehat{\text{P}}$ and P be a set of promising PNPs contained by P. We inductively define a sequence of locations $\pi = q^0 q^1 \ldots$ such that all the following conditions hold:

1. $q^0 \in Q_0$.
2. $(q^i, x, q^{i+1}) \in \Delta$ holds for every $i \in \mathbb{N}$ and for some element $x \in \Sigma_{\mathcal{M}}$.

3. $\sigma[i..] \not\approx \widehat{q^i_{Old}} \wedge \circ\widehat{q^i_{Next}}$ holds for every $i \in \mathbb{N}$.

4. for every promising PNP P_0 in P there exists some $i \in \mathbb{N}$ such that there exists some PNP P_1 contained by q^i_{Old} such that P_1 is a fulfilling PNP of P_0.

As induction base we choose some $q \in Q_0$ as q^0 such that $\widehat{\mathrm{P}} \longleftrightarrow \widehat{q_{Old}} \wedge \circ\widehat{q_{Next}}$ holds. The existence of such location is ensured by Lemma 4.16. By assumption $\sigma \not\approx \widehat{\mathrm{P}}$ which implies that $\sigma[0..] \not\approx \widehat{\mathrm{P}}$. It follows that $\sigma[0..] \not\approx \widehat{q^0_{Old}} \wedge \circ\widehat{q^0_{Next}}$.
Moreover, for every promising PNP P_0 in P, if q_0 contains some PNP P_1 such that P_1 is one of the fulfilling PNPs of P_0, we remove P_1 from P.
Now assume that we have already defined a sequence of locations $q^0q^1 \dots q^k$ such that condition 1 holds, condition 2 holds for every $i \in 0..k-1$ and condition 3 holds for every $i \in 0..k$ and there exists. We consider the following cases:

- If condition 4 holds or $P \neq \emptyset$. Then by Lemma 4.5, there exists a transition $(q^k, x, q) \in \Delta$ such that $\sigma[k+1..] \not\approx \widehat{q_{Old}} \wedge \circ\widehat{q_{Next}}$ for some element $x \in \Sigma_{\mathcal{M}}$. Choose such a location q as q^{k+1}.

- If condition 4 doesn't hold and there exists some transition $(q^{j-1}, x, q) \in \Delta$ such that for some promising PNP P_0 in P there exists some PNP P_1 contained by q_{Old} such that P_1 is a fulfilling PNP of P_0. Choose such a location q as q^{k+1}. Remove P_0 from P.

Notice that since $\sigma \not\approx \widehat{\mathrm{P}}$ holds by assumption, the existence of such location stated in the second case of induction step is ensured by Lemma 4.15. Thus we can construct π which is accepting as required. ■

For the illustration of the formula graphs and the automaton for formulas $v, \circ v, p \rightarrow q, \Box p, \neg\Box p, \Box[p]_v$ and $\neg\Box[p]_v$ see Figure A.1-A.7 in Appendix A.

We also observe that every SCS of the constructed automaton is reachable as stated in the following lemma. This lemma will be used later in proving the completeness of predicate diagrams in next chapter.

Lemma 4.19 *Every* SCS S *in* $\mathcal{M}$ *is reachable.*

Proof. Every node is reachable, by Construction 4.10 every location is reachable. ■

4.4 Timed automata

In the following is a brief introduction of timed automata taken from [7, 8, 9] and [32].

A *timed automaton* is a finite automaton augmented with a finite set of real-valued *clocks*. We assume that transitions are instantaneous. However, time can elapse when the automaton is in a location. When a transition occurs, some of the clocks may be reset to zero. At any instant, the reading of a clock is equal to the time that has elapsed since the last time the clock was reset. We assume that time passes at the same rate for all clocks.

A clock constraint, called a *guard*, is associated with each transition. The transition can be taken only if the current values of the clocks satisfy the clock constraint. A clock constraint is also associated with each location of the automaton. This constraint is called the *invariant* of the location. Time can elapse in the location only as long as the invariant of the location is true. An example of a timed automaton is shown in Figure 4.8 [32]. The automaton consists of two locations q_0 and q_1, two clocks x and y, an "a" transition from q_0 to q_1 and a "b" transition from q_1 to q_0. The automaton starts in location q_0. It can remain in that location as long as the clock y is less than or equal to 5. As soon as the value y is greater than or equal to 3, the automaton can make an "a" transition to location q_1 and reset the clock y to 0. The automaton can remain in location q_1 as long as y is less than or equal to 10 and x is less than or equal to 8. When y is at least 4 and x is at least 6, it can make a "b" transition back to location q_0 and reset x.

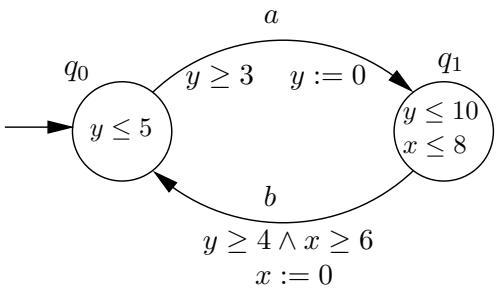

Figure 4.8: A simple timed automaton.

We now define timed automata formally.

Definition 4.20 ***(time sequence and timed word)*** *A time sequence $\varphi = \varphi_0\varphi_1 \ldots$ is an infinite sequence of time values $\varphi_i \in \mathbb{R}^+$, satisfying the following constraints:*

1. $\varphi_0 = 0$.

2. *Monotonicity: τ increases strictly monotonically; that is, $\varphi_i < \varphi_{i+1}$ for all $i \in \mathbb{N}$.*

3. *Progress: For every $t \in \mathbb{R}^+$ there is some $i \in \mathbb{N}$ such that $\varphi_i > t$.*

A timed word (w, φ) is a pair where $w \in \Sigma^\omega$ is an infinite word over Σ and φ is a time sequence.

We need to say what type of clock constraints are allowed on the edges.

Definition 4.21 ***(clock constraint)*** *Let X be a set of clock variables, ranging over the nonnegative real number $\mathbb{R}^+$. The set of clock constraints $\Theta(X)$ is defined as follows:*

- *All inequalities of the form $x \prec c$ or $c \prec x$ are in $\Theta(X)$, where $\prec$ is either $<$ or $\leq$ and c is nonnegative rational number.*
- *If θ_1 and θ_2 are in $\Theta(X)$ then $\theta_1 \wedge \theta_2$ is in $\Theta(X)$.*

Note that if X contains k clocks, then each clock constraint is a *convex* subset of k-dimensional Euclidean space. Thus, if two points satisfy a clock constraint, then all of the points on the line segment connecting these points satisfy the clock constraint.

A *clock* interpretation ν for a set X of clocks assigns a real value to each clock; that is, it is a mapping from X to $\mathbb{R}^+$. For $t \in \mathbb{R}^+$, $\nu + t$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + t$. For $Y \subseteq X, \nu[Y := 0]$ denotes the clock interpretation for X which assigns 0 to each $x \in Y$, and agrees with ν over the rest of the clocks.

Definition 4.22 ***(timed automata)*** *A timed automaton Γ over an alphabet Σ is a tuple $(Q, Q_0, \mathcal{X}, \mathcal{I}, \Lambda)$, where*

- *Q is a set of locations,*
- *$Q_0 \subseteq Q$ is a set of initial locations,*
- *$\mathcal{X}$ is a finite set of clocks,*
- *$\mathcal{I} : Q \to \Theta(\mathcal{X})$ is a mapping from locations to clock constrains, called the location invariant, and*

- $\Lambda \subseteq Q \times \Sigma \times \Theta(\mathcal{X}) \times 2^{\mathcal{X}} \times Q$ *is a set of transitions. The 5-tuple* $(q, a, \theta, \lambda, q')$ *corresponds to a transition from location* q *to location* q' *labelled with* a*, a constraint* θ *that specifies when the transition is enabled and a set of clocks* $\lambda \subseteq \mathcal{X}$ *that are reset when the transition is executed.*

Given a timed word (w, φ), the timed automaton Γ starts in one of its start locations at time 0 with all its clocks initialized to 0. As time advances, the values of all clocks change, reflecting the elapsed time. At time φ_i, Γ changes location from q to q' using some transition of the form $\langle q, a, \theta, \lambda, q' \rangle$ reading the input a, if the current values of the clocks satisfies θ. With this transition the clocks in λ are reset to 0, and thus start counting time with respect to the time of occurrence of this transition. This behavior is captured by defining runs of timed automata. A run records the location and the values of all the clocks at the transition points.

Definition 4.23 ***(run of a timed automaton)*** *Let* $\Gamma = (Q, Q_0, \mathcal{X}, \mathcal{I}, \Lambda)$ *be a timed automaton over* Σ*. A run of* Γ*,* ϕ*, is an infinite sequence of the form:*

$$\phi : (q_0, \nu_0) \xrightarrow{w_1, \varphi_1} (q_1, \nu_1) \xrightarrow{w_2, \varphi_2} \ldots$$

where $q_i \in Q$*,* $\nu_i \in [\mathcal{X} \to \mathbb{R}^+]$*,* $w_i \in \Sigma$ *and* $\varphi_i \in \mathbb{R}^+$ *such that the following conditions hold:*

- $q_0 \in Q_0$ *and* $\nu_0(x) = 0$ *for all* $x \in \mathcal{X}$*.*
- *For all* $i \in \mathbb{N}$*, there is an edge in* Λ*,* $(q_i, w_i, \lambda_i, \theta_i, q_{i+1})$*, such that* $(\nu_{i+1} + \varphi_{i+1} - \varphi_i)$ *satisfies* θ_{i+1}*,* ν_{i+1} *is equal to* $[\lambda_i \to 0](\nu_i + \varphi_{i+1} - \varphi_i)$ *and for every* $0 \leq e \leq \varphi_{i+1} - \varphi_i$*, the invariant* $\mathcal{I}(q_i)$ *holds.*

The set $inf(\phi)$ *consists of those locations* $q \in Q$ *such that* $q = q_i$ *for infinitely many* $i \geq 0$*.*

4.5 Discussion and related work

There are some other ω-automata over infinite words whose structures are the same as Muller automata but differ in the definition of accepting conditions. Büchi automata [24] is the simplest class of ω-automata over infinite

words.The accepting condition of these automata is a set of locations and a run is accepting iff $\inf(\pi) \cap \mathcal{F} \neq \emptyset$, that is, when some accepting state appears in π infinitely often. Generalized Büchi automata differ from the standard Büchi by allowing multiple accepting sets rather than only one. An accepting run needs to pass through each of one of the sets in $\mathcal{F}$ infinitely often. Another automata is Street automata whose accepting condition is a set of pairs (E, F) where E and F are sets of locations. A run π is accepting if

$$\bigwedge_{i=1}^{n} (\mathrm{Inf}(\pi) \cap E_i \neq \emptyset \vee \mathrm{Inf}(\pi) \cap F_i = \emptyset).$$

This accepting condition represents a "fairness condition" which can be read as "for each i, if some location of F_i is visited infinitely often, then some location of E_i is visited infinitely often".

There are many algorithm that can be used for translating (propositional) linear temporal logic formulas into automata, as proposed by Danielle et al.[39], Somenzi&Bloem [101], Gastin&Oddoux [49], Wolper [106] and many others. Most of them use (generalized) Büchi automata as the target automata.

The algorithm for constructing Muller automaton is inspired by the one for construction generalized Büchi automata from LTL formulas [51, 32, 93]. This algorithm can be used for model-checking of a temporal formula. This algorithm is called "on-the-fly" construction, meaning that the construction of the automaton is done in the same time with the property checking. Thus, whenever a violation of the checked property is discovered, the construction can be stopped before generating the entire automaton.

Since pTLA* is a sub-logic of PTL (Propositional Temporal Logic), in fact, there is no real need to define a special automaton construction for pTLA* in order to check satisfiability. The reason for defining automata construction here is to support the clarification of the completeness proof of predicate diagrams in the next chapter.

The concept of timed automata will be used in Chapter 6 where we talk about the verification of real-time systems.

Chapter 5

Discrete systems

5.1 Overview

This chapter deals with the first class of reactive systems concerned in this thesis, namely discrete systems.

We begin with the specification of discrete systems. The specification we use here is a restricted form of the general TLA specification described in Chapter 3 by requiring the next-state relation formula be written as a disjunction of some action formulas. Another restriction is that the liveness property is expressed by fairness property over sub-actions of the next-state relation formula.

In the next section we describe a class of diagrams that will be used to verify this class of reactive systems. Predicate diagrams, first introduced by Cansell, Méry and Merz in [26], is a class of diagrams intended as the basis for the verification of both safety and liveness properties of reactive systems. We also describe how the verification of temporal properties of discrete systems to be done using predicate diagrams.

As illustration we take the Bakery Algorithm from Lamport and prove some properties of this algorithm using predicate diagrams.

The main contribution of this chapter is the completeness proof of predicate diagrams, which is given in Section 5.6. We show that predicate diagram is complete, i.e. for any specification and any formula of the temporal propositional logic, if the specification implies the formula, then the implication can be proven by a suitable predicate diagram.

In the end of this chapter, we discuss our approach and compare to some other work.

5.2 Specification

In this work we represent the specification of discrete systems as formula of the form:

$$Spec \equiv Init \wedge \Box[Next]_v \wedge L_f \tag{5.1}$$

where

- *Init* and v as defined in Formula 3.1,
- *Next* is a disjunction of actions representing the next-state relation of the system and
- L_f is a conjunction of formula $\mathrm{WF}_v(A)$ and $\mathrm{SF}_v(A)$ where A is any action such that $Spec \rightarrow \Box[A \rightarrow Next]_v$ holds.

Formula 5.1 is a restricted form of Formula 3.1. The liveness property L_f is expressed as a conjunction of formulas of the form $\mathrm{WF}_v(A)$ and/or $\mathrm{SF}_v(A)$ where A is an action that implies *Next* in all states that are reachable for *Spec*. We will call such action A sub-action of *Next*. This form ensures the machine-closedness of the specification as stated in the following theorem [1].

Theorem 5.1 *If* Π *is a safety property and* L *is the conjunction of a finite or countably infinite number of formulas of the form* $\mathrm{WF}_v(A)$ *and/or* $\mathrm{SF}_v(A)$ *such that each* $\langle A \rangle_v$ *is a sub-action of* Π*, then* (Π, L) *is machine closed.*

5.3 Predicate diagrams

Now we present a class of diagrams that will be used for the verification of discrete systems.

The underlying assertion language, by assumption, contains a finite set $\mathcal{O}$ of binary relation symbols $\prec$ that are interpreted by well-founded orderings. For $\prec \in \mathcal{O}$, its reflexive closure is denoted by $\preceq$. We write $\mathcal{O}^=$ to denote the set of relation symbols $\prec$ and $\preceq$ for $\prec$ in $\mathcal{O}$.

A *predicate diagram* is a finite graph whose nodes are labeled with sets of (possibly negated) predicates, and whose edges are labeled with actions (more precisely, action names) as well as optional annotations that assert certain expressions to decrease with respect to an ordering in $\mathcal{O}^=$. Intuitively, a node of a predicate diagram represents the set of system states that satisfy the formulas contained in the node. (We indifferently write n for the set and the conjunction of its elements.) An edge (n, m) is labeled with action A if A can

cause a transition from a state represented by n to a state represented by m. An action A may have an associated fairness condition; fairness conditions apply to all transitions labeled by the action rather than to individual edges.

Formally, the definition of predicate diagram is relative to finite sets $\mathcal{P}$ and $\mathcal{A}$ that contain the state predicates and the (names of) actions of interest; we will later use $\tau \notin \mathcal{A}$ to denote a special stuttering action. We write $\overline{\mathcal{P}}$ to denote the set of literals formed by the predicates in $\mathcal{P}$, that is, the union of $\mathcal{P}$ and the negations of the predicates in $\mathcal{P}$.

Definition 5.2 *(predicate diagram) Assume given two finite sets $\mathcal{P}$ and $\mathcal{A}$ of state predicates and action names. A predicate diagram $G = (N, I, \delta, o, \zeta)$ over $\mathcal{P}$ and $\mathcal{A}$ consists of*

- *a finite set $N \subseteq 2^{\overline{\mathcal{P}}}$ of nodes,*
- *a finite set $I \subseteq N$ of initial nodes,*
- *a family $\delta = (\delta_A)_{A \in \mathcal{A}}$ of relations $\delta_A \subseteq N \times N$; we also denote by δ the union of the relations δ_A, for $A \in \mathcal{A}$ and write $\delta_=$ to denote the reflexive closure of the union of these relations,*
- *an edge labeling o that associates a finite set $\{(t_1, \prec_1), \ldots, (t_k, \prec_k)\}$, of terms t_i paired with a relation $\prec_i \in \mathcal{O}^=$ with every edge $(n, m) \in \delta$, and*
- *a mapping $\zeta : \mathcal{A} \to \{\mathrm{NF}, \mathrm{WF}, \mathrm{SF}\}$ that associates a fairness condition with every action in $\mathcal{A}$; the possible values represent no fairness, weak fairness, and strong fairness.*

We say that the action $A \in \mathcal{A}$ can be taken at node $n \in N$ iff $(n, m) \in \delta_A$ holds for some $m \in N$, and denote by $En(A) \subseteq N$ the set of nodes where A can be taken. We say that the action $A \in \mathcal{A}$ can be taken along an edge (n, m) iff $(n, m) \in \delta_A$.

We now define *runs* and *traces* through a diagram as the set of those behaviors that correspond to fair runs satisfying the node and edge labels. To evaluate the fairness conditions we identify the enabling condition of an action $A \in \mathcal{A}$ with the existence of A-labeled edges at a given node. For a term x and two states s and t, we denote by $s[\![x]\!]$ and $s[\![x]\!]t$ for $s \models\!\approx x$ and $s, t \models\!\approx x$, respectively.

Definition 5.3 *(run, trace) Let $G = (N, I, \delta, o, \zeta)$ be a predicate diagram over sets $\mathcal{P}$ and $\mathcal{A}$. A run of G is an ω-sequence $\rho = (s_0, n_0, A_0)\ (s_1, n_1, A_1)\ldots$ of triples where s_i is a state, $n_i \in N$ is a node and $A_i \in \mathcal{A} \cup \{\tau\}$ is an action such that all of the following conditions hold:*

1. *$n_0 \in I$ is an initial node.*
2. *$s_i[\![n_i]\!]$ holds for all $i \in \mathbb{N}$.*
3. *For all $i \in \mathbb{N}$, either $A_i = \tau$ and $n_i = n_{i+1}$ or $A_i \in \mathcal{A}$ and $(n_i, n_{i+1}) \in \delta_{A_i}$.*
4. *If $A_i \in \mathcal{A}$ and $(t, \prec) \in o(n_i, n_{i+1})$, then $s_{i+1}[\![t]\!] \prec s_i[\![t]\!]$.*
5. *If $A_i = \tau$ then $s_{i+1}[\![t]\!] \preceq s_i[\![t]\!]$ holds whenever $(t, \prec) \in o(n_i, m)$ for some $m \in N$.*
6. *For every action $A \in \mathcal{A}$ such that $\zeta(A) = \mathrm{WF}$ there are infinitely many $i \in \mathbb{N}$ such that either $A_i = A$ or $n_i \notin En(A)$.*
7. *For every action $A \in \mathcal{A}$ such that $\zeta(A) = \mathrm{SF}$, either $A_i = A$ holds for infinitely many $i \in \mathbb{N}$ or $n_i \in En(A)$ holds for only finitely many $i \in \mathbb{N}$.*

We write $runs(G)$ to denote the set of runs of G.

The set $tr(G)$ of traces through G consists of all behaviors $\sigma = s_0 s_1 \ldots$ such that there exists a run $\rho = (s_0, n_0, A_0)\ (s_1, n_1, A_1)\ldots$ of G based on the states in σ.

Informally, $\sigma = s_0 s_1 \ldots$ is a trace through the predicate diagram G if we can find a sequence of nodes n_i whose associated formulas are true at s_i and that are related by transitions whose edge labels, including the ordering annotations, are satisfied by consecutive states. In addition to the transitions that are explicitly represented by edges of the diagram, we allow stuttering transitions that remain in the source node. This definition ensures that the set of traces through a diagram is closed under stuttering equivalence, just as the models of TLA* formulas. Note that we do not require that two states s_i and s_{i+1} related by stuttering step (i.e. $A_i = \tau$) be identical, they are only required to satisfy the same node label. However, if node n_i has an outgoing edge with an ordering annotation $(t, \prec)$ stuttering transitions are forbidden to increase the value of t, as otherwise stuttering could interfere with liveness properties induced by well-founded orderings.

Fairness conditions are used to prevent infinite stuttering. Their interpretation is standard, based on the intuition that the enabledness of actions with non-trivial fairness requirements is reflected in the diagram.

5.4 Verification

In this section we describe the verification of discrete systems using predicate diagrams.

In linear-time formalisms such as TLA and TLA*, trace inclusion is the appropriate implementation relation. Thus, a specification *Spec* implements a property or high level specification F if and only if the implication $Spec \rightarrow F$ is valid [69]. Predicate diagrams can be used to refine this implication into two conditions: first, all behaviors allowed by *Spec* must also be traces through the diagram and second, every trace through the diagram must satisfy F. Although both conditions are stated in terms of trace inclusion, following Cansell et al., two different techniques are used here. To show that a predicate diagram is a correct representation of a specification, we consider the node and edge labels of the diagram as predicates on the concrete state space of *Spec*, and reduce trace inclusion to a set of proof obligations that concern individual states and transitions. On the other hand, to show that the diagram implies the high level property, we regard all labels as Boolean variables. The predicate diagram can therefore be encoded as a finite labeled transition system, whose temporal properties are established by model checking. In this respect, predicate diagrams act as an interface between interactive and automatic proof methods. We now consider both aspects in more detail.

5.4.1 Conformance

When comparing a specification and a predicate diagram, we must first assign meaning to the action names that appear in the diagram. We assume given a function α that assigns an action formula to every action name. Because no confusion is possible, we will leave this assignment implicit, and again write A instead of $\alpha(A)$ when referring to the formula assigned to the name A.

A predicate diagram G is said to conform to a specification *Spec*, written $Spec \trianglelefteq G$, if every behavior that satisfies *Spec* is a trace through G. In general, proving conformance requires reasoning about entire behaviors. The following theorem essentially introduces a set of first-order ("local") verification conditions that are sufficient to establish conformance of a diagram to

a discrete system specification in standard form (Formula 5.1).

Theorem 5.4 *(conformance) Let $G = (N, I, \delta, o, \zeta)$ be a predicate diagram over $\mathcal{P}$ and $\mathcal{A}$, and Spec $\equiv$ Init $\wedge \Box[Next]_v \wedge L_f$ be a discrete system specification. G conforms to Spec if all of the following conditions hold:*

1. $\models Init \rightarrow \bigvee_{n \in I} n$.
2. $\approx n \wedge [Next]_v \rightarrow n' \vee \bigvee_{(A,m):(n,m)\in\delta_A} \langle A \rangle_v \wedge m'$ *holds for every node* $n \in N$.
3. *For all* $n, m \in N$ *and all* $(t, \prec) \in o(n, m)$:
   (a) $\approx n \wedge m' \wedge \bigvee_{A:(n,m)\in\delta_A} \langle A \rangle_v \rightarrow t' \prec t$ *and*
   (b) $\approx m \wedge [Next]_v \wedge n' \rightarrow t' \preceq t$.
4. *For every action* $A \in \mathcal{A}$ *such that* $\zeta(A) \neq \mathrm{NF}$:
   (a) *If* $\zeta(A) = \mathrm{WF}$ *then* $\models Spec \rightarrow \mathrm{WF}_v(A)$.
   (b) *If* $\zeta(A) = \mathrm{SF}$ *then* $\models Spec \rightarrow \mathrm{SF}_v(A)$.
   (c) $\models n \rightarrow$ ENABLED $\langle A \rangle_v$ *holds whenever* $n \in En(A)$.
   (d) $\approx n \wedge \langle A \rangle_v \rightarrow \neg m'$ *holds for all* $n, m \in N$ *such that* $(n, m) \notin \delta_A$.

Condition 1 asserts that every initial state of the system must be covered by some initial node. This ensures that every run of the system can start at some initial node of the diagram. Condition 2 asserts that from every node, every transition, if it is enabled then it must have a place to go, i.e., there is a successor node which represents the successor state of the transition. It proves that every run of the system can stay in the diagram. Condition 3 is related to the ordering annotations whereas Condition 4 is related to the fairness conditions. For the proof of this theorem the readers may refer to [26, 83].

5.4.2 Model checking predicate diagrams

For the proof that all traces through a predicate diagram satisfy some property F we view the predicate diagram as a finite transition system that is amenable to model checking. All predicates and actions that appear as labels of nodes or edges are then viewed as atomic propositions.

Regarding predicate diagrams as finite labeled transition systems, their runs can be encoded in the input language of standard model checkers such as SPIN [57]. Two variables indicate the current node and the last action taken. The predicates in $\mathcal{P}$ are represented by boolean variables, which are updated according to the label of the current node, nondeterministically, if that label contains neither P nor $\neg P$. We also add variables $b_{(t,\prec)}$, for every term t and relation $\prec \in \mathcal{O}$ such that $(t, \prec)$ appears in some ordering annotation $o(n, m)$. These variables are set to 2 if the last transition taken is labeled by $(t, \prec)$, to 1 if it is labeled by $(t, \preceq)$ or is stuttering transition and to 0 otherwise.

The fairness conditions associated with the actions of a diagram are easily expressed as LTL assumptions for SPIN. As in Definition 5.3 we assume that action A is enabled whenever the currently active node has an outgoing edge in δ_A. To capture the effect of the ordering annotations, we add Streett-type formulas[1] $\Box\Diamond(b_{(t,\prec)} = 2) \rightarrow \Box\Diamond(b_{(b,\prec)} = 0)$ as additional assumptions for every variable $b_{(t,\prec)}$ introduced. These assumptions ensure that transitions known to strictly decrease t with respect to $\prec$ can not be taken infinitely often unless infinitely often some transitions are taken that may increase the value of t.

In order to establish the properties F whose atomic formulas are contained in the set $\mathcal{P}$ of predicates of interest, one can now model check the transition system resulted from the encoding. If the verification succeeds then every trace through the diagram satisfies F, and by transitivity of trace inclusion it follows that F holds on any specification that conforms to the diagram. The counter-examples produced by the model-checker, on the other hand, need not to correspond to actual system runs, because detail has been lost in the abstraction. Such counter-examples, nevertheless, are helpful in order to refine the abstraction, for example by adding ordering annotations. Obviously, the size of diagrams that can be effectively analyzed in this way is mostly limited by the number of fairness conditions and ordering annotations present in the diagram.

5.5 An example: Bakery algorithm

In this section we will illustrate the use of predicate diagrams by proving some properties of LAMPORT's Bakery algorithm for mutual exclusion [67].

The LAMPORT's Bakery algorithm is one of the famous solutions to the mutual exclusion problem for the general case of N process. The algorithm is based on the idea of a ticket machine, where people entering a (big) bakery

[1] See Section 4.5.

draw a ticket with a number on it that indicates their turn to buy their Sunday morning croissants.

We now consider a version of the algorithm for two processes [83] based on atomic actions[2]. The Pseudocode of the protocol is given in Figure 5.1, where angle brackets denote instantaneous atomic actions. Variables t_1 and t_2 indicate the ticket number of each process. Each process can be in five control locations, $l_0, \ldots, l_4$ for process P_1 and $m_0, \ldots, m_4$ for process P_2. We will call the control locations l_0 and m_0 non-critical sections, l_1 and m_1 requesting sections, l_2 and m_2 trying sections, l_3 and m_3 critical-sections and l_4 and m_4 exiting sections.

First, every process is in its non-critical section and then it moves to its requesting section. Process P_i draws a ticket by setting its own number t_i to the number of the other process incremented by one, while moving from the requesting to the trying section. Process P_i then stays at its trying section until it is allowed to enter its critical section. Finally, process P_i leaves the critical section by resetting its corresponding ticket number t_i to zero and moves to its exiting section.

We wish to establish the mutual-exclusion property as well as the liveness properties of the algorithm, that is, we want to prove that it is never the case that both processes are in their critical sectiona at the same time (mutually exclusive access of the clients to croissants) and that every process will eventually be in its critical section (eventual access, once having a ticket).

integer $t_1, t_2 = 0$;
cobegin

loop		**loop**
l_0 : noncritical;		m_0 : noncritical
l_1 : $\langle t_1 := t_2 + 1 \rangle$;		m_1 : $\langle t_2 := t_1 + 1 \rangle$;
l_2 : **await** $\langle t_2 = 0 \vee t_1 \leq t_2 \rangle$;	$\|\|$	m_2 : **await** $\langle t_1 = 0 \vee \neg(t_1 \leq t_2) \rangle$;
l_3 : critical section;		m_3 : critical section;
l_4 : $\langle t_1 := 0 \rangle$		m_4 : $\langle t_2 := 0 \rangle$
endloop		**endloop**

coend

Figure 5.1: Bakery algorithm for two processes: Pseudocode representation.

The specification is given in Figure 5.2. Notice that in the specification

[2]The original version of Bakery algorithm is non-atomic. We have chosen the atomic version because it generates fewer proof obligations.

module BAKERY

Specification

$$
\begin{array}{rcl}
Init & \equiv & pc_1 = 0 \wedge pc_2 = 0 \wedge t_1 = 0 \wedge t_2 = 0 \\
NCrit_1 & \equiv & pc_1 = 0 \wedge pc_1' = 1 \wedge \text{UNCHANGED } \langle pc_2, t_1, t_2 \rangle \\
Req_1 & \equiv & pc_1 = 1 \wedge pc_1' = 2 \wedge t_1' = t_2 + 1 \wedge \text{UNCHANGED } \langle pc_2, t_2 \rangle \\
Try_1 & \equiv & \wedge\ pc_1 = 2 \wedge (t_2 = 0 \vee t_1 \leq t_2) \\
 & & \wedge\ pc_1' = 3 \wedge \text{UNCHANGED } \langle pc_2, t_1, t_2 \rangle \\
Crit_1 & \equiv & pc_1 = 3 \wedge pc_1' = 4 \wedge \text{UNCHANGED } \langle pc_2, t_1, t_2 \rangle \\
Exit_1 & \equiv & pc_1 = 4 \wedge pc_1' = 0 \wedge t_1' = 0 \wedge \text{UNCHANGED } \langle pc_2, t_2 \rangle \\
NCrit_2 & \equiv & pc_2 = 0 \wedge pc_2' = 1 \wedge \text{UNCHANGED } \langle pc_1, t_1, t_2 \rangle \\
Req_2 & \equiv & pc_2 = 1 \wedge pc_2' = 2 \wedge t_2' = t_1 + 1 \wedge \text{UNCHANGED } \langle pc_1, t_1 \rangle \\
Try_2 & \equiv & \wedge\ pc_2 = 2 \wedge (t_1 = 0 \vee \neg(t_1 \leq t_2)) \\
 & & \wedge\ pc_2' = 3 \wedge \text{UNCHANGED } \langle pc_1, t_1, t_2 \rangle \\
Crit_2 & \equiv & pc_2 = 3 \wedge pc_2' = 4 \wedge \text{UNCHANGED } \langle pc_1, t_1, t_2 \rangle \\
Exit_2 & \equiv & pc_2 = 4 \wedge pc_2' = 0 \wedge t_2' = 0 \wedge \text{UNCHANGED } \langle pc_1, t_1 \rangle \\
Next & \equiv & \vee\ NCrit_1 \vee Req_1 \vee Try_1 \vee Crit_1 \vee Exit_1 \\
 & & \vee\ NCrit_2 \vee Req_2 \vee Try_2 \vee Crit_2 \vee Exit_2 \\
vars & \equiv & \langle pc_1, pc_2, t_1, t_2 \rangle \\
Bakery & \equiv & Init \wedge \Box[Next]_{vars} \wedge \mathrm{WF}_{vars}(Next)
\end{array}
$$

Theorem

1. $Bakery \rightarrow \Box\neg(pc_1 = 3 \wedge pc_2 = 3)$
2. $Bakery \rightarrow \Box(pc_1 = 2 \rightarrow \Diamond pc_1 = 3)$
3. $Bakery \rightarrow \Box(pc_2 = 2 \rightarrow \Diamond pc_2 = 3)$

Figure 5.2: Module BAKERY.

we use variable pc_1 and pc_2 whose values ranging over 0..4 for representing the control locations, i.e. $l_0..l_4$ and $m_0..m_4$ for process 1 and 2, respectively. The properties to be verified are stated in the **Theorem** part.

Figure 5.3 shows the suitable predicate diagram for the Bakery algorithm. Using Theorem 5.4.1 we can show that the predicate diagram in Figure 5.3 conforms to the specification BAKERY in Figure 5.2. For example, we have:

- $Init \rightarrow pc_1 = 0 \wedge pc_2 = 0 \wedge t_1 = 0 \wedge t_2 = 0 \wedge t_1 \leq t_2$
- $$\begin{pmatrix} pc_1 = 1 \wedge \\ pc_2 = 1 \wedge \\ t_1 = 0 \wedge \\ t_2 = 0 \wedge \\ t_1 \leq t_2 \end{pmatrix} \wedge [Next]_v \rightarrow \begin{pmatrix} pc_1 = 1 \wedge \\ pc_2 = 1 \wedge \\ t_1 = 0 \wedge \\ t_2 = 0 \wedge \\ t_1 \leq t_2 \end{pmatrix} \vee \begin{pmatrix} pc_1 = 2 \wedge \\ pc_2 = 1 \wedge \\ t_1 > 0 \wedge \\ t_2 = 0 \wedge \\ \neg(t_1 \leq t_2) \end{pmatrix}$$

Encoding the predicate diagram in Promela, the input language of SPIN, as described in Section 5.4.2, and then model-checking the resulted transition

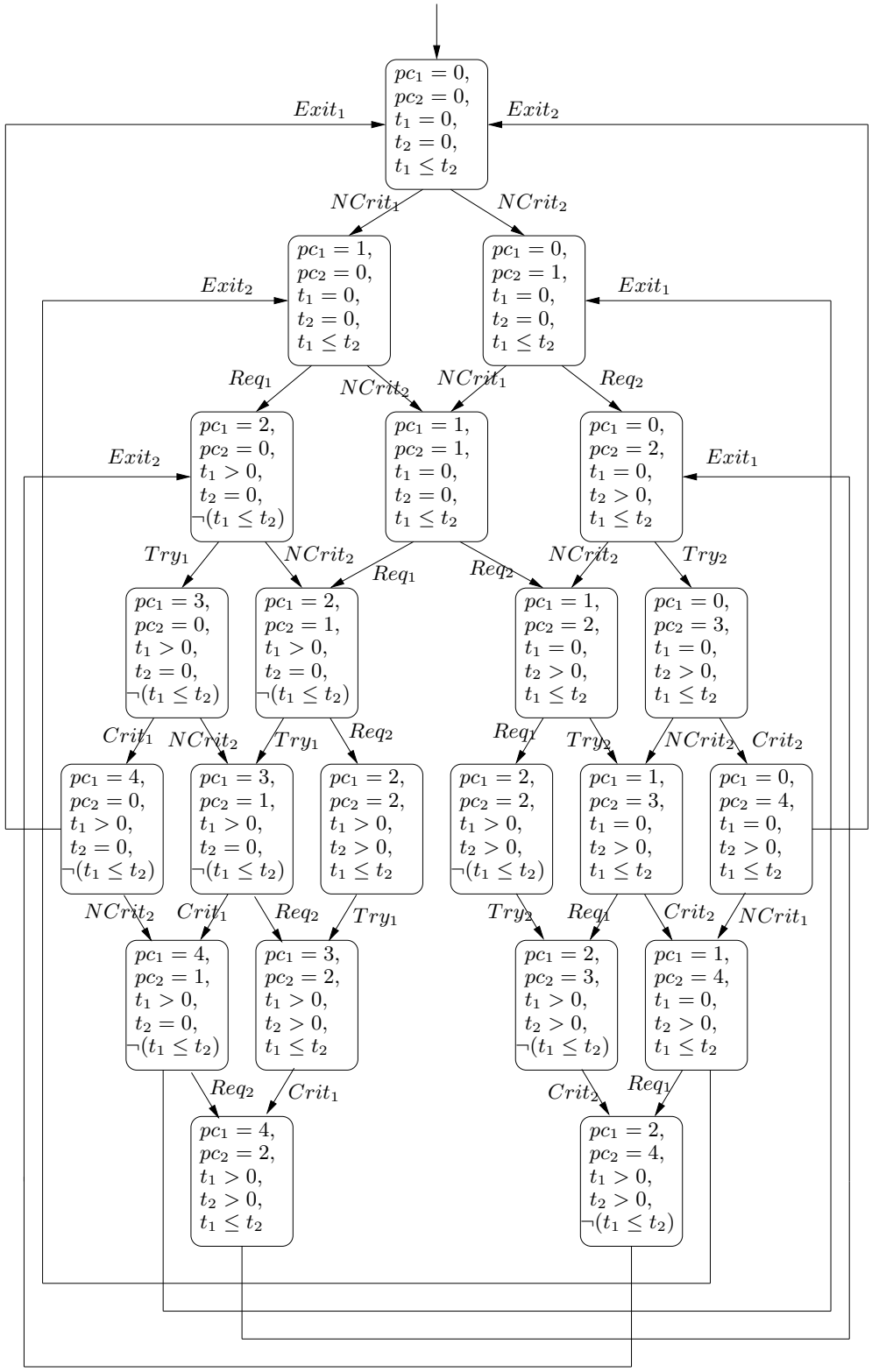

Figure 5.3: Predicate diagram for Bakery algorithm.

system using SPIN, we can verify the mutual exclusion and eventual access properties of Bakery algorithm.

5.6 Completeness of predicate diagrams

Proving the completeness of predicate diagrams means to show that for any specification and any formula of the temporal propositional logic (in this case we take pTLA*), if the specification implies the formula, then the implication can be proven by a suitable predicate diagram.

Theorem 5.5 *For any specification, Spec* $\equiv$ *Init* $\wedge \Box[Next]_v \wedge L_f$ *and any formula of pTLA*, F, if* $\models$ *Spec* $\rightarrow$ *F holds, then there exists a predicate diagram* $G = (N, I, \delta, o, \zeta)$ *such that Spec* $\trianglelefteq$ *G and* $tr(G) \subseteq F$ *hold.*

Before we prove the main theorem we present some definitions and supporting statements.

Definition 5.6 *Given a run through a predicate diagram,* $\rho = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$. *We define* $\inf(\rho)$ *to be the set of nodes which occur infinitely often in* ρ.

Given a predicate diagram, a specification and a node of the diagram, we assume that if some condition hold then we can determine the states that are reachable by the specification that hold at that node.

Assumption 5.7 *(accessible states 1) Let* $G = (N, I, \delta, o, \zeta)$ *be a predicate diagram over* $\mathcal{P}$ *and* $\mathcal{A}$ *and Spec = Init* $\wedge \Box[Next]_v \wedge L_f$ *be a specification. Let* Σ *be a set of states that are reachable by Spec. Then there exists a mapping* acc1 *from nodes to predicates, such that for every state in* Σ *and for every node in N,* $s[\![\text{acc1}(n)]\!]$ *iff there exists a model prefix* $s_0 s_1 \ldots s_k$ *of Spec and a run prefix through* $G, \rho = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots(s_k, n_k, A_k)$ *such that* $s_k = s$ *and* $n_k = n$. *That is,* $\text{acc1}(n)$ *represents the accessible states at node* n.

In this following proof, we partially rely on the concept of SCS for representing the loops of the systems. We recall SCS as a set of nodes such that for every pair of nodes there exists a path connecting them that visits only nodes in the set.

A SCS is called *Spec*-reachable if there exists some model of *Spec* such that its corresponding run through G visits every node in this SCS. Using the Assumption 5.7, *Spec*-reachable SCS can be defined as follows.

Definition 5.8 *(Spec-reachable* SCS*) Let* $G = (N, I, \delta, o, \zeta)$ *over* $\mathcal{P}$ *and* $\mathcal{A}$ *be a predicate diagram,* $Spec = Init \wedge \Box[Next]_v \wedge L_f$ *be a specification and* $S = \{n_1, \ldots, n_k\}$ *be a* SCS *of* G*. Then* S *is called a Spec-reachable* SCS *if for every* $i \in 1..k$, $\mathrm{acc1}(n_i) \neq \emptyset$.

It is clear that every *Spec*-reachable SCS is also a reachable SCS of G, i.e., for every node n in the SCS, there exists a path from some initial node in I to n.

Generally, SCSs in the diagram correspond to loops in the system. Since the objective of the diagram is to approximate a set of computations as precisely as possible, we want to exclude from the diagram those sequences of states that are not computations of the system.

Definition 5.9 *(accepting run) Let* $G = (N, I, \delta, o, \zeta)$ *be a predicate diagram over* $\mathcal{P}$ *and* $\mathcal{A}$ *and* $Spec \equiv Init \wedge \Box[Next]_v \wedge L_f$ *be a specification. Then a run through* $G, \rho = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ *is called an accepting run if* $s_0 s_1 \ldots \models Spec$.

A SCS is called accepting if there is some accepting run through a predicate diagram that visits every node of the SCS infinitely often.

Definition 5.10 *(accepting* SCS*) Let* $G = (N, I, \delta, o, \zeta)$ *over* $\mathcal{P}$ *and* $\mathcal{A}$ *be a predicate diagram,* $S = \{m_1, \ldots m_k\}$ *be a* SCS *of* G *and* $Spec = Init \wedge \Box[Next]_v \wedge L_f$ *be a specification.* S *is called accepting if there exists some accepting run through* G, $\rho = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ *such that* $\inf(\rho) = S$*. Otherwise, we call it terminating* SCS.

If a SCS is not *Spec*-reachable then it is clear that it is also terminating. We want in particular to exclude the loops of the systems represented by *Spec*-reachable but terminating SCSs by using the ordering annotation. A SCS S is well-founded if there exists some edge labeled with $(t, \prec)$ and every other edge is labeled with either $(t, \prec)$ or $(t, \preceq)$, for a term t and a binary relation $\preceq \in \mathcal{O}^{=}$, such that the value of t will not increase along edges labeled by $(t, \preceq)$ and will decrease along edges labeled by $(t, \prec)$ whenever this loop is traversed by some run through G, i.e. there is no run through G which can stay forever in this loop due to the well-foundedness of the domain.

Definition 5.11 *(well-founded* SCS*) Let* $G = (N, I, \delta, o, \zeta)$ *over* $\mathcal{P}$ *and* $\mathcal{A}$ *be a predicate diagram and* $S = \{m_1, \ldots, m_k\}$ *be a* SCS *of* G*.* S *is called well-founded if there exists a pair of term* t *and a symbol* $\prec \in \mathcal{O}$*, such that* $\{(t, \prec), (t, \preceq)\} \cap o(m_i, m_j) \neq \emptyset$ *holds for every edge* $m_i, m_j \in S$ *and there exists some edge* (m_i, m_j) *in* S *such that* $(t, \prec) \in o(m_i, m_j)$.

In the case of well-founded SCS, there is no run through G which can visit every node in this SCS infinitely often due to the well-foundedness of the domain. Thus, it is obvious that every well-founded SCS is also a terminating SCS.

Lemma 5.12 *Let*

- $G = (N, I, \delta, o, \zeta)$ *over* $\mathcal{P}$ *and* $\mathcal{A}$ *be a predicate diagram,*
- $Spec = Init \wedge \Box[Next]_v \wedge L_f$ *be a specification,*
- $S = \{m_1, \ldots, m_k\}$ *be a Spec-reachable but terminating* SCS*,*
- $Z = \{(m, s) : m \in S, s[\![\text{acc1}(m)]\!]\}$ *be a set containing pairs of node in* S *and state in* Σ *that is reachable at node* n*, and*
- $\prec$ *be an ordering over* Z *such that* $(m_a, s_a) \succ (m_b, s_b)$ *iff there exists some run segment,* $(s_i, n_i, A_i) \ldots (s_j, n_j, A_j)$*, such that* $(n_i, s_i) = (m_a, s_a)$ *and* $(n_j, s_j) = (m_b, s_b)$ *that traverses some edge in* S*.*

Then $\prec$ *is a well-founded ordering over* Z*.*

Proof: Let G, S, Z and $\prec$ as defined. We will show that $\prec$ is a well-founded ordering over Z.

- $\preceq$ is partial order. It is clear reflexive and transitive. To show that it is also antisymmetric, suppose that $(m_a, s_a) \succeq (m_b, s_b), (m_b, s_b) \succeq (m_a, s_a)$ and $(m_a, s_a) \neq (m_b, s_b)$. Let $(s_0, n_0, A_0)(s_1, n_1, A_1) \ldots (s_a, n_a, A_a)$ be a run prefix through G. Then we can continue this sequence and get $(s_0, n_0, A_0)(s_1, n_1, A_1) \ldots ((s_a, n_a, A_a) \ldots (s_b, n_b, A_b) \ldots (s_{a-1}, n_{a-1}, A_{a-1}))^\omega$, where n_{a-1} is the predecessor of n_a in the path from n_b to n_a, that traverses every edge in S infinitely often, a contradiction.
- $\prec$ is well-founded. Suppose there exists a sequence of elements of Z, $(n_1, s_1)(n_2, s_2) \ldots$ such that $(n_1, s_1) \succ (n_2, s_2) \succ \ldots$. Then we can construct an accepting run through G that traverses every node in S infinitely often, a contradiction. ■

The following assumption will be needed later for determining the ordering annotation of the predicate diagram.

Assumption 5.13 *(existence of ordering annotation) We assume that our language is sufficiently expressive to encode a suitable term such that for every* SCS S *in* G*, if it is Spec-reachable but terminating* SCS *then we can find a pair of term* t *and a binary symbol* $\prec \in \mathcal{O}^=$ *for the suitable ordering annotation for* S*.*

Corollary 5.14 *Every Spec-reachable but terminating* SCS *is well-founded.*

We also define the so-called fair-exit of a SCS which is some action A such that $\zeta(A) = \text{WF}$ such that A can be taken at every node in S but cannot be taken along every edge in S or if there exists some action A with $\zeta(A) = \text{SF}$ such that A enabled at some node in S but cannot be taken along every edge in S.

Definition 5.15 *(fair-exit) Let $G = (N, I, \delta, o, \zeta)$ over $\mathcal{P}$ and $\mathcal{A}$ be a predicate diagram and $S = \{m_1, \ldots, m_k\}$ be a* SCS *of G.*

1. *An action $A \in \mathcal{A}$ such that $\zeta(A) = \text{WF}$ is called a weak-exit for S if*
   - *$(m_i, m_j) \notin \delta_A$ holds for every edge $m_i, m_j \in S$ and*
   - *$m_i \in En(A)$ holds for every node $m_i \in S$.*
2. *An action $A \in \mathcal{A}$ such that $\zeta(A) = \text{SF}$ is called a strong-exit for S if*
   - *$(m_i, m_j) \notin \delta_A$ holds for every edge $m_i, m_j \in S$ and*
   - *there exists some node $m_i \in S$ such that $(m_i, n) \in En(A)$ holds for some node $n \in N \setminus S$.*

Let $Spec \equiv Init \wedge \Box[Next]_v \wedge L_f$ be a system specification and F be a pTLA* formula such that $\models Spec \rightarrow F$ holds. We construct a predicate diagram G such that $Spec \trianglelefteq G$ and $tr(G) \subseteq F$ hold.

We will construct three Muller automata in this proof. First, we construct a Muller automaton $\mathcal{M}^f = (Q^f, Q_0^f, \Delta^f, \mathcal{F}^f)$ for formula F using Construction 4.10. We call this automaton *formula automaton.*

Lemma 5.16 *(properties of $\mathcal{M}^f$)*

1. *Let $\sigma = s_0 s_1 \ldots$ be a behavior. Then σ is a model of F iff there exists an accepting run of $\mathcal{M}^f$, $\pi = q^0 q^1 \ldots$, such that $\sigma[i..] \approx \widehat{q^i_{Old}} \wedge \circ\widehat{q^i_{Next}}$ holds for every $i \in \mathbb{N}$.*
2. *Every* SCS *in $\mathcal{M}^f$ is reachable.*

Proof: Analogous to the proof of Theorem 4.11 and Lemma 4.19. ■

Next, we define some sets and functions as follows:

Definition 5.17 *Let* $Spec \equiv Init \wedge \Box[Next]_v \wedge L_f$ *be a specification. We define some sets and functions as follow:*

1. $\mathcal{A}$ *is a set containing Next and every action name that appears in* L_f*.*
2. $\zeta : A \to \{\mathrm{NF}, \mathrm{WF}, \mathrm{SF}\}$ *is a function mapping every action* $A \in \mathcal{A}$ *to* $\{\mathrm{NF}, \mathrm{WF}, \mathrm{SF}\}$ *which is defined by*

$$\zeta(A) = \begin{cases} \text{WF whenever } \models Spec \to \mathrm{WF}_v(A). \\ \text{SF whenever } \models Spec \to \mathrm{SF}_v(A). \\ \text{NF, otherwise.} \end{cases}$$

As before, it is assumed that there exists a function α that assigns an action formula to every action name A in $\mathcal{A}$. We will leave this assignment implicit and again write A instead of $\alpha(A)$ when referring to the formula assigned to the name A.

Based on Definition 5.17 we construct a Muller automaton over the alphabet $\mathcal{A} \cup \{\tau\}$ using the following construction.

Construction 5.18 *Let* $\mathcal{M}^s = (Q^s, Q_0^s, \Delta^s, \mathcal{F}^s)$ *be a Muller automaton over* $\mathcal{A}$ *where*

1. $Q^s = Q_0^s = \{(T, B) : T \subseteq \mathcal{A} \text{ and } B \in \mathcal{A} \cup \{\tau\}\}$
2. Δ^s *is defined by* $((T_1, A_1), B, (T_2, A_2)) \in \Delta^s$ *iff* $B \in T_1$ *and* $B = A_2$ *or* $B = \tau$ *and* $(T_1, A_1) = (T_2, A_2)$*.*
3. $\mathcal{F}^s$ *is defined by*

$$\mathcal{F}^s = \bigcap_{A \in \mathcal{A}: \zeta(A) \neq \mathrm{NF}} F_A$$

where F_A *is a set of* SCS*s and*

- *If* $\zeta(A) = \mathrm{WF}$ *then for every* $F \in F_A, F$ *contains some location* (T_1, A_1) *such that* $A \notin T_1$ *or there exists an edge* $((T_1, A_1), (T_2, A_2))$ *in* F *such that* $A_2 = A$*.*
- *If* $\zeta(A) = \mathrm{SF}$ *then for every* $F \in F_A, A \notin T_1$ *holds for every location* (T_1, A_1) *in* F *or there exists an edge* $((T_1, A_1), (T_2, A_2))$ *in* F *such that* $A_2 = A$*.*

The intended meaning of a location $(T, A) \in Q^s$ is that if $T \neq \emptyset$ then for every action A_1 in T, $\langle A_1 \rangle_v$ is enabled on the location and for every action A_2 which is not in T, $\langle A_2 \rangle_v$ is not enabled on the location, and that the location has been reached by taking action $\langle A \rangle_v$ if $A \neq \tau$ or by taking a stuttering transition; and if $T = \emptyset$ then there is no action in $\mathcal{A}$ which is enabled on this location and the only transition which is enabled is the stuttering transition τ. The accepting condition is defined in a way such that it exactly characterizes the fairness of *Spec*. It includes only those SCSs in which every fair transition is either taken or not enabled on some location in the SCS for every action A such that $\zeta(A) = \text{WF}$, or is either taken or not enabled on any location in the SCS for every action A such that $\zeta(A) = \text{SF}$.

We call the second automaton *specification automaton*. The first property of the constructed Muller automaton is stated by the following lemma.

Lemma 5.19 *Let $\sigma = s_0 s_1 \ldots$ be a model of Spec. Then there exists an accepting run $\pi = (T_0, A_0), (T_1, A_1) \ldots$ such that $s_i [\![\langle A_{i+1} \rangle_v]\!] s_{i+1}$ holds for every $i \in \mathbb{N}$.*

Proof. Let $\sigma = s_0 s_1 \ldots$ be a model of *Spec*. We inductively define a sequence of locations $\pi = (T_0, A_0)(T_1, A_1) \ldots$ such that

1. $(T_0, A_0) \in Q_0^s$,
2. For every $i \in \mathbb{N}$:
   (a) $T_i = \{A \in \mathcal{A} : s_i [\![\text{ENABLED } \langle A \rangle_v]\!]\}$,
   (b) if $T_i = \emptyset$ then $A_{i+1} = \tau$ else $A_{i+1} \in T_i$ holds for every $A \in T_i$,
   (c) $((T_i, A_i), A_{i+1}, (T_{i+1}, A_{i+1})) \in \Delta^s$ and
   (d) if $A_{i+1} \in \mathcal{A}$ then $s_i [\![\langle A_{i+1} \rangle_v]\!] s_{i+1}$.

For the induction base, we choose a location $(T_0, A_0) \in Q_s^0$ such that $T_0 = \{A \in \mathcal{A} : s_0 [\![\text{ENABLED } \langle A \rangle_v]\!]\}$. The existence of such location is ensured by the definition of the locations in the Construction 5.18.

For the induction step, assume that we already have a sequence of locations $(T_0, A_0) \ldots (T_k, A_k)$ such that the conditions 1 and 2a hold for every $i \in 0..k$ and conditions 2b, 2c and 2d hold for every $i \in 0..k-1$. Choose some location $(T, A) \in Q^s$ such that if $T_k = \emptyset$ then $T = \emptyset$ and $A = \tau$ and otherwise $T = \{B \in \mathcal{A} : s_{k+1} [\![\text{ENABLED } \langle B \rangle_v]\!]\}$, $A \in T_k$ and $s_k [\![\langle A \rangle_v]\!] s_{k+1}$. The existence of such location is ensured by the definition of the transition in the Construction 5.18.

To complete the proof, assume that there exists an action $A \in \mathcal{A}$ such that $\models \mathit{Spec} \rightarrow \text{WF}(A)$ and there exists some $i \in \mathbb{N}$ such that $s_j [\![\text{ENABLED } \langle A \rangle_v]\!]$ holds for every $j \geq i$. Then by Definition 5.17, $\zeta(A) = \text{WF}$. We show that π

is accepting. From Construction 5.18 and the definition of accepting run, π is accepting if (T_i, A) holds for infinitely many $i \in \mathbb{N}$. Since $\sigma \models \mathit{Spec}$ holds by assumption, it follows that $s_i[\![\langle A \rangle_v]\!]s_{i+1}$ holds for infinitely many $i \in \mathbb{N}$ and by condition 2c, $A_i = A$ holds for infinitely many $i \in \mathbb{N}$ as required.

For action $A \in \mathcal{A}$ such that $\models \mathit{Spec} \rightarrow \mathrm{SF}_v(A)$, the proof is analogous, replacing the assumption with "there exists an action $A \in \mathcal{A}$ such that $\models \mathit{Spec} \rightarrow \mathrm{SF}_v(A)$ and $s_i[\![\textsc{enabled}\ \langle A \rangle_v]\!]$ holds for infinitely many $i \in \mathbb{N}$". ■

The second property of the automaton resulted from Construction 5.18 is related to the definition of accepting conditions.

Lemma 5.20 *Let S be a* SCS *in $\mathcal{M}^s$ such that $S \notin \mathcal{F}^s$. Then there exists an outgoing edge $((T_1, A_1), (T_2, A))$ such that $\zeta(A) \neq \mathrm{NF}$.*

Proof. Let S be an SCS in $\mathcal{M}^s$ such that $S \notin \mathcal{F}^s$. Then, from the construction, there exists some action $A \in \mathcal{A}$ such that $\zeta(A) \neq \mathrm{NF}$, such that $A_j \neq A$ holds for every edge $((T_i, A_i), (T_j, A_j))$ in S, and if $\zeta(A) = \mathrm{WF}$ then $A \in T_i$ holds for every location (T_i, A_i) in S and if $\zeta(A) = \mathrm{SF}$ then $A \in T_i$ holds for some location (T_i, A_i) in S. By the construction of the edges, any location on which a transition is enabled has outgoing edges for the transition, and therefore in the case of $\zeta(A) = \mathrm{WF}$, $((T_i, A_i), (T_j, A))$ holds for every location (T_i, A_i) in S and in the case of $\zeta(A) = \mathrm{SF}$, $((T_i, A_i), (T_j, A))$ holds for every location (T_i, A_i) in S such that $A \in T_i$. ■

Next, we construct an automaton which is the *product automaton* of $\mathcal{M}^f$ and $\mathcal{M}^s$. Since the input alphabet of $\mathcal{M}^f$ and $\mathcal{M}^s$ are different we cannot use the standard product automaton construction, Construction 4.2. In this case we take $\mathcal{A} \cup \{\tau\}$ which is the input alphabet of $\mathcal{M}^s$ as the input alphabet.

Construction 5.21 *The product automaton $\mathcal{M}^p$ of $\mathcal{M}^f$ and $\mathcal{M}^s$ over alphabet $\mathcal{A} \cup \{\tau\}$ is a tuple $(Q^p, Q^p{}_0, \Delta^p, \mathcal{F}^p)$ where Q^p, Q^p_0 and $\mathcal{F}^p$ are given by:*

- $Q^p = Q^f \times Q^s$,
- $Q^p_0 = Q^f_0 \times Q^s_0$,
- $\mathcal{F}^p = \mathcal{F}^p_f \cap \mathcal{F}^p_s$ *where* $\mathcal{F}^p_f = \mathcal{F}^f \times Q^s$ *and* $\mathcal{F}^p_s = Q^f \times \mathcal{F}^s$;

whereas Δ^p is defined in a way such that $((q^i, (T_i, A_i)), A, (q^{i+1}, (T_{i+1}, A_{i+1})) \in \Delta^p$ iff

- $(q^i, x, q^{i+1}) \in \Delta^f$,

- $((T_i, A_i), A, (T_{i+1}, A_{i+1})) \in \Delta^s$ *and*
- $\models \widehat{q^i_{Old}} \rightarrow$ ENABLED $\langle A \rangle_v$.

Notice that the definition of the transition relation ensures that Δ^p contains only relations that appears in $\mathcal{M}^f$ and $\mathcal{M}^s$.

Some important properties of $\mathcal{M}^p$ which will be used later are stated in the following lemma.

Lemma 5.22

1. *Let $\sigma = s_0 s_1 \ldots$ be a model of Spec $\wedge$ F. Then there exists an accepting run $\pi = (q^0, (T_0, A_0))(q^1, (T_1, A_1)) \ldots$ such that $\sigma[i..] \models \widehat{q^i_{Old}} \wedge \circ\widehat{q^i_{Next}}$ and $s_i [\![\langle A \rangle_v]\!] s_{i+1}$ holds for every $i \in \mathbb{N}$.*
2. *Let $\sigma = s_0 s_1 \ldots$ be a model of Spec. If $\models$ Spec $\rightarrow F$ then there exists an accepting run $\pi = (q^0, (T_0, A_0))\ (q^1, (T_1, A_1)) \ldots$ such that $s_i [\![\langle A_{i+1} \rangle_v]\!] s_{i+1}$ holds for every $i \in \mathbb{N}$.*
3. *Let S be a* SCS *in $\mathcal{M}^p$ such that $S \notin \mathcal{F}^p_s$. Then there exists an outgoing edge $((q^1, (T_1, A_1)), (q^2, (T_2, A_2)))$ such that $\zeta(A_2) \neq$ NF.*
4. *For every* SCS *S in $\mathcal{M}^p$, for every location q in S, there exists some path from some initial location $q^0 \in Q^p_0$ to n.*

Proof: (sketch)

1. It follows from Lemma 5.16(1), Lemma 5.19 and Construction 5.21.
2. It follows from the assumption that *Spec* implies F, Lemma 5.19 and Construction 5.21.
3. It follows from Lemma 5.20 and Construction 5.21.
4. By Lemma 4.19 every SCS of $\mathcal{M}^p$ is reachable. By Construction 5.18 every location of $\mathcal{M}^s$ is an initial location. Thus Construction 5.21 ensures that every SCS of $\mathcal{M}^p$ is reachable. ■

The next step is the construction of predicate diagram that is expected to be the suitable predicate diagram we are looking for. We use the following assumption for defining the nodes of the diagrams. This assumption is similar to Assumption 5.7, only now we consider the relation between specification and product automaton.

Assumption 5.23 *(accessible state 2) Let Spec = Init* $\wedge \Box[Next]_v \wedge L_f$ *be a specification and* $\mathcal{M}^p$ *be the product automaton resulted from Construction 5.21. Let* Σ *be a set of states that are reachable by Spec. Then there exists a mapping* acc2 *from locations to predicates such that* $s[\![\text{acc2}((q,(T,A))]\!]$ *iff there exists a model prefix* $s_0 s_1 \ldots s_k$ *of Spec and a run prefix* $\pi = (q^0,(T_0,A_0))(q^1,(T_1,A_1))\ldots(q^k,(T_k,A_k))$ *of* $\mathcal{M}$ *such that for every* $i \in 0..k$, $s_i[\![\widehat{q^i_{Old}}]\!]$ *and* $s_i[\![\text{ENABLED}\ \langle B\rangle_v]\!]$ *holds for every* $B \in T_i$. *That is,* acc2$((q,(T,A))$ *represents the accessible states at location* $(q,(T,A))$.

Construction 5.24 *(predicate diagram) Let* $\mathcal{M}^p = (Q^p, Q^p_0, \Delta^p, \mathcal{F}^p)$ *be the product automaton from Construction 4.2. We define a predicate diagram* $G = (N, I, \delta, o, \zeta)$ *as follows:*

- *For every location* $(q,(T,A))$, *G contains a node labeled by* acc2$((q,(T,A))$ *in N. Moreover, this node is in I iff* $(q,(T,A)) \in Q^p_0$.
- *For every* $A \in \mathcal{A}$ *and every* n, m *in N,* $(n,m) \in \delta_A$ *iff there exists a transition* $((q_1,(T_1,A_1)),(x,A_2),(q_2,(T_2,A_2))) \in \Delta^p$ *such that* $n =$ acc2$((q_1,(T_1,A_1)))$, $m =$ acc2$((q_2,(T_2,A_2)))$ *and* $A_2 = A$.
- $o = \emptyset$.
- ζ *as defined in Definition 5.17.*

We can prove that every model of *Spec* is a trace through this diagram but we still cannot guarantee that every trace through the diagram is a model of F. We should exclude the runs through the diagram whose corresponding traces are not the models of *Spec* by making some SCSs in G terminating.

Definition 5.25 *(corresponding* SCS*) For a* SCS $S = \{n_1,\ldots,n_k\}$ *in G, we call a* SCS *in* $\mathcal{M}^p$, $S' = \{(q^1,(T_1,A_1)),\ldots,(q^k,(T_k,A_k))\}$ *the corresponding* SCS *of S if* $n_i =$ acc2$((q^i,(T_i,A_i)))$ *holds for* $i \in 1..k$.

A SCS S of G is terminating if the corresponding SCS S' in $\mathcal{M}^p$ is not accepting, i.e. $S' \notin \mathcal{F}^p$ or $S' \notin \mathcal{F}^p_f \cap \mathcal{F}^p_s$.

Lemma 5.26 *Let S be a* SCS *in G and S′ be its corresponding* SCS *in* $\mathcal{M}^p$. *If* $S' \notin \mathcal{F}^p_s$ *then S is a fair-exit* SCS.

Proof. By Lemma 5.22 (3) for every SCS S' such that $S' \notin \mathcal{F}^p_s$, there exists an out-going edge $((q_1,(T_1,A_1)),(q_2,(T_2,A_2)))$ such that $\zeta(A_2) \neq$ NF. Since Construction 5.24 does not change the underlying graph of the automaton, the resulted predicate diagram has also this property and by Definition 5.15, A_2 now become the fair-exit for S. ■

Lemma 5.27 *Let S be a* SCS *in G and S' be its corresponding* SCS *in $\mathcal{M}^p$. If $S' \notin \mathcal{F}^p_f$ then S is a Spec-reachable* SCS.

Proof. By Lemma 5.22, inherits from the property of $\mathcal{M}^f$, every SCS of $\mathcal{M}^p$ is reachable. Since by assumption $Spec \rightarrow F$, S is also a *Spec*-reachable SCS. ■

Since terminating SCS of G is *Spec*-reachable, by Assumption 5.13 we can define the ordering annotation of G.

It remains to prove the two following lemmas to complete the proof of Theorem 5.5. Notice we do not use Conformance 5.4 to prove that G conforms to *Spec*.

Lemma 5.28 *Every model of Spec is a trace through G.*

Proof. Let $\sigma = s_0 s_1 \ldots$ be a behavior such that $\sigma \models Spec$. By Lemma 5.22 (2), there exists an accepting run of $\mathcal{M}^p$, $\pi = (q^0, (T_0, A_0))(q^1, (T_1, A_1)) \ldots$ such that $s_i [\![\langle A \rangle_{i+1}]\!] s_{i+1}$ holds for every $i \in \mathbb{N}$. We define a run through $G, \rho = (s_0, \mathrm{acc2}((q^0, (T_0, A_0))), A_1), (s_1, \mathrm{acc2}((q^1, (T_1, A_1))), A_2) \ldots$. The existence of such a run is ensured by the Construction 5.24, since it doesn't change the underlying graph. Let S be $\inf(\rho)$ and let $S' = \inf(\pi)$ be the corresponding SCS of S. Since π accepting, $S' \in \mathcal{M}^p$. By the definition of accepting condition in Construction 5.18 and the definition of ordering annotation of G, S is an accepting SCS. ■

Lemma 5.29 *Every trace through G satisfies F.*

Proof. Let $\sigma = s_0 s_1 \ldots$ be a trace through G, $\rho = (s_0, n_0, A_0)(s_1, n_1, A_1) \ldots$ be the corresponding run of σ, and S be a SCS of G such that $\inf(\rho) = S$. Assume that $\sigma = s_0 s_1 \ldots \not\models F$. Let S' be the corresponding SCS of S in $\mathcal{M}^p$. By assumption $\sigma \not\models F$, implies that $S \notin \mathcal{F}^p$. If $S' \notin \mathcal{F}^p_f$, then by Lemma 5.27 S is a *Spec*-reachable SCS. Moreover, since S' is not accepting, S is terminating. By Corollary 5.14 S is well-founded. If $S' \notin \mathcal{F}^p_s$, then by Lemma 5.26, S has a fair-exit. But then, by assumption $\inf(\rho) = S$, contradiction. ■

5.7 Discussion and related work

We have presented a method for the verification of discrete systems. Discrete systems are represented as TLA* formulas. The verification is done by using predicate diagrams. Predicate diagrams integrate the deductive and algorithmic verification techniques. In our approach, the verification is done in two steps. The first step is to find a predicate diagram that conforms

to the specification. This conformance is proven deductively by proving the proof obligations as stated in Theorem 5.4. The second step is regarding the predicate diagram as a finite labeled transition system which amenable for model-checking. We have applied our approach on the Bakery algorithm.

We show that predicate diagram is complete, i.e. for any specification and any formula of the temporal propositional logic, if the specification implies the formula, then the implication can be proven by a suitable predicate diagram. Given a specification *Spec* and a pTLA* formula F such that *Spec* satisfies F, we have proven the completeness by generating a predicate diagram that conforms *Spec* and satisfies F. In proving these completeness we construct three Muller automata: the formula automaton $\mathcal{M}^f$, the specification automaton $\mathcal{M}^s$ and the product automaton $\mathcal{M}^p$. The formula automaton is automaton which accepts exactly the behaviors satisfying F. We used the construction described in Chapter 4. Based on the information in the specification, in particular the actions contained in *Spec*, we construct the specification automaton such that the accepting condition is defined in a way such that it exactly characterizes the fairness of *Spec*. The specification automaton has properties that it conforms to *Spec*. The third automaton is the product automaton of $\mathcal{M}^f$ and $\mathcal{M}^s$. Thus, the properties of $\mathcal{M}^p$ are inherited from $\mathcal{M}^f$ and $\mathcal{M}^s$. The last step is translate this product automaton into the final predicate diagram.

The first work using graphs to visualize and structure temporal proofs for concurrent programs is due to Owicki and Lamport [90]. There, proof lattices are used to better explain logic rules and to "see" and so verify what a program is supposed to do. Building on these ideas, *proof charts* were introduced in Cousot's thesis [35]. A proof chart for a transition system is a finite graph with a unique start and final state. Proof obligations are associated with every sub-chart, and "return" edges can be labeled by well-founded orderings to guarantee terminating of the system. In contrast to our diagram, these approaches concentrate on illustrating the structure of the proof rather than of the system under analysis.

Predicate-action diagrams proposed by Lamport [70] represent the safety part of specifications. An interesting point is that different, complementary views about a specification can be illustrated by different diagrams and that the proof of refinement relations via diagrams is considered.

Manna et al. have also advocated the diagrammatic verification of temporal logic properties [19]. *Verification Diagrams*, introduced by Manna & Pnueli [79], provide graphical representation of the direct proof of temporal properties. Verification diagrams are dedicated to particular classes of

properties: *Invariance* diagrams which used to prove invariance properties, *Wait-for* diagrams for precedence properties and *Chain* and *Rank* diagrams that can be used to prove response properties. *Generalized Verification Diagrams* [20, 99] extend the framework to arbitrary temporal formulas. *Modular Verification Diagrams* [21] allow the combination of several generalized verification diagrams into a single proof. The combined set represents the intersection of the languages described by each diagram. *Fairness diagram* [41] represents the possible system states and transitions: the progress and response properties of the system are encoded by *fairness constraints* which generalize the usual notions of fairness. Given a system and a temporal specification, a proof begins with an initial fairness diagram that directly corresponds to the system. This diagram is then transformed into one which corresponds directly to the specification, or which can be shown to satisfy it by purely algorithmic methods.

The main difference of their approach to our diagram is in the representation of fairness conditions, which we assert on the level of entire diagrams rather than for individual edges. This simplifies the verification conditions related to fairness assumptions. We also allow an arbitrary number of ordering annotations, which reduces the number of proof obligations, and should be more intuitive for a system designer.

In our proof, we rely on the expressiveness of our language in encoding such ordering whenever some conditions hold. Although some methods for defining such ordering systematically have been proposed, for example the work from Dams et al. [38], practically the ordering is still defined manually and intuitively.

The completeness proof here is inspired by the completeness proof of *generalized verification diagrams*[99], in particular the use of Muller automata for proving the completeness.

The formal translation of predicate diagrams into SPIN code is given in [83].

Chapter 6

Real time systems

6.1 Overview

Computers are frequently used in critical applications where predictable response times are essential for correctness. Examples of such applications include controllers for aircraft, industrial machinery and robots. Such systems, that capture the metric aspects of time, are called real-time systems. This chapter is devoted to the specification and verification of real-time systems.

First, we give the standard formula for real-time specification we use here. A real-time program can be written as the conjunction of its untimed version, expressed in a standard way as a TLA* formula, and its timing assumptions, expressed in terms of a few standard parameterized formulas. This form is, again, a restricted form of the general specification described in Chapter 3. The separation between specification of untiming and timing properties makes real-time specification easier to write and to understand.

In the next section, we define a variant of predicate diagrams, which we call timed predicate diagrams that can be used to verify real-time systems. A timed predicate diagram can be viewed as a predicate diagram equipped with some components in order to constraint the time.

The verification of real-time systems using timed predicate diagrams will be given section 6.4. As illustration, we take the FISCHER's protocol problem, which will be given in Section 6.5.

To conclude this chapter, we give some related work and discuss our approach.

6.2 Specification

We now describe the real-time systems specification. In the whole discussion we denote by ∞ a value that is greater than any real number.

We follow the general format of TLA real-time specification suggested by ABADI & LAMPORT in [1] where real time is modeled by a non-negative real-valued variable *now*. We assume that initially *now* is equal to 0, and it increases monotonically and without bound, which excludes "Zeno" behaviors[1]. Because it is convenient to make time-advancing steps distinct from ordinary program steps, the tuple of the system's (discrete) variables, v, should not change when *now* advances. This condition can be expressed by the *time-progress* formula $RTNow(v)$, which is defined as follows:

$$\begin{aligned} RTNow(v) \quad \equiv \quad & \wedge\ now \in \mathbb{R}^+ \\ & \wedge\ \Box[now' \in \{r \in \mathbb{R}^+ : r > now\} \wedge \text{UNCHANGED } v]_{now} \\ & \wedge\ \forall r \in \mathbb{R}^+ :<> (now > r) \end{aligned} \tag{6.1}$$

We express real-time constraints by placing timing bounds on actions. Such an action on which we place the timing bound is called a *time-bounded action*. For any time-bounded action A, we associate a real variable t which we call the *corresponding timer* of A. We assume that neither t nor *now* appear in A. We now define a formula $Timer(t)$ for asserting that t always equals the length of time that $\langle A \rangle_v$ has been continuously enabled since the last $\langle A \rangle_v$ step. The value of t should be set to 0 by an $\langle A \rangle_v$ step or a step that disables $\langle A \rangle_v$. A step that advances *now* should increment t by $now' - now$ iff $\langle A \rangle_v$ is enabled. Changes to t are therefore described by the action:

$$\begin{aligned} TNext(A, v, t) \equiv t' = \textbf{if } \langle A \rangle_v \vee \neg(\text{ENABLED } \langle A \rangle_v)' \ \textbf{then } & 0 \\ \textbf{else } & t + (now' - now). \end{aligned} \tag{6.2}$$

The meaning of $Timer(t)$ is interesting only when v is a tuple whose components include all the variables that may appear in A. In this case, a step that leaves v unchanged cannot enabled or disabled $\langle A \rangle_v$. The formula $Timer(t)$, therefore, should allow steps that leave t, v, and *now* unchanged. We let initial value of t be 0,

[1] A behavior such that time keeps advancing but is bounded by some limit.

$$TInit(t) \equiv t = 0, \tag{6.3}$$

and define

$$Timer(A, v, t) \equiv TInit(t) \wedge \Box[TNext(A, v, t)]_{\langle t, v, now \rangle}. \tag{6.4}$$

Basically, the specification of real-time systems are similar to the one given by Formula 3.1. They differ only on the way we govern when the transitions may, or must, be taken. In real-time systems, we should state explicitly those conditions in the term of time unit. Since, it is conventional to express timing assumptions by a mix of *lower* and *upper bound*, we introduce two real numbers d and e, where $0 \leq d \leq e \leq \infty$, to govern when $\langle A \rangle_v$ transition may, or must, be taken:

- $\langle A \rangle_v$ can be taken if it has been continuously enabled for at least d seconds since the last $\langle A \rangle_v$ step - or since the beginning of the behavior. This property is expressed by the formula:

$$MinTime(A, v, t, d) \equiv \Box[A \rightarrow (t \geq d)]_v. \tag{6.5}$$

  We call d the *lower bound* of t.

- $\langle A \rangle_v$ can be continuously enabled for at most e seconds before an $\langle A \rangle_v$ step must occur. This property is expressed by the formula:

$$MaxTime(t, e) \equiv \Box(t \leq e). \tag{6.6}$$

  We call e the *upper bound* of t.

We now define the so-called *real-time bound* formula which will be associated to every bounded-time action in our specification:

$$\begin{array}{lll} RTBound(A, v, t, d, e) & \equiv & \wedge\ Timer(A, v, t) \\ & & \wedge\ MaxTime(t, e) \\ & & \wedge\ MinTime(A, v, t, d). \end{array} \tag{6.7}$$

The specification of real-time systems now can be written as formula of the form:

$$RTSpec \equiv Init \wedge \Box[Next]_v \wedge RTNow(v) \wedge RT \tag{6.8}$$

where $Init, Next$ and v are as defined in Formula 3.1, $RTNow(v)$ is the formula defined in Formula (6.1), and RT is a conjunction of real-time bound formulas $RTBound(A_i, v, t_i, d_i, e_i)$ where A_i is a sub-action or disjunct of $Next$.

In Figure 6.1, we give a small example of a real-time system. This system consists of two process, Up and $Down$ and a shared variable x. Initially x is set to some natural number. Process Up keeps incrementing x; whereas process $Down$ is responsible to set x to 0 whenever x is greater than 0. We put a time constraint on the process $Down$, such that x must be reset to 0 in not more than 3 seconds after x becomes greater than 0. We want to prove that always eventually x is equal to 0.

module LOOP

Specification

$$\begin{aligned} Init &\equiv x \in \mathbb{N} \\ Up &\equiv x' = x + 1 \\ Down &\equiv x > 0 \wedge x' = 0 \\ Loop &\equiv Init \wedge \Box[Up \vee Down]_x \wedge RTNow(x) \\ &\qquad \wedge RTBound(Down, x, t, 0, 3) \end{aligned}$$

Theorem

1. $Loop \rightarrow \Box\Diamond(x = 0)$

Figure 6.1: Module LOOP.

6.3 Timed predicate diagrams

We now present *timed predicate diagrams*, or TPDs for short, which are a variant of predicate diagrams described in the previous section. The idea of these diagrams is to use the components of predicate diagrams related to the discrete properties and to replace the components related to the fairness conditions with some components related to real-time conditions.

For the components related to real-time property, we adopt the structure of timed-automata (see Section 4.4). A TPD is equipped with a finite set of

real-valued variables that measure time. These variables are called *timers*. Every timer is associated with some predicates over it, which we call *time-constraints*.

Definition 6.1 *(time-constraint) A time-constraint is a state predicate of the form $c\#z$ or $c_1 - c_2\#z$ where c, c_1 and c_2 are timers, $\# \in \{\leq, <, =, >, \geq\}$ and z is a real constant, including ∞.*

For a set of timers C, we denote by Φ_C and $\psi(\Phi_C)$, the set of time-constraints over timers $c \in C$ and the set containing all $c \in C$ that appears in Φ_C, respectively.

We now give the formal definition of TPDs.

Definition 6.2 *(TPD) Given a set of state predicates $\mathcal{P}$, a set of actions $\mathcal{A}$, a set of timers C and a set of time-constraints over the timers in C, Φ_C, TPD T over $\mathcal{P}, \mathcal{A}, C$ and Φ_C is given by a tuple $(N, I, \delta, o, r, g, R)$ where*

- *N, I, δ and o as defined in Definition 5.1.*
- *A mapping $r : N \rightarrow 2^{\Phi_C}$ that associates a set of time-constraints in Φ_C with every node in N.*
- *A mapping $g : N \times N \rightarrow 2^{\Phi_C}$ that associates a set of time-constraints in Φ_C with every edge in δ.*
- *A mapping $R : N \times N \rightarrow 2^C$ that associates a set of timers in C with every edge in δ.*

We say that the action $A \in \mathcal{A}$ can be taken at node $n \in N$ iff $(n, m) \in \delta_A$ holds for some $m \in N$, and denote by $En(A) \subseteq N$ the set of nodes where A can be taken. We say that the action $A \in \mathcal{A}$ can be taken along (n, m) iff $(n, m) \in \delta_A$.
A timer $c \in C$ is called an active timer on a node $n \in N$ if there exists some node $m \in N$ such that $g(n, m)$ contains some time-constraint over c. We denote by $act(n)$ the set of active timers on n.

Like predicate diagrams, TPD can be viewed as a labeled directed graph. Different from predicate diagrams, the nodes of TPDs may be labeled with one more set of predicates which we call *time-invariant*. This invariant and the state predicates over system's discrete variables must be satisfied on every node.

The edges of TPDs may be labeled with actions and time-constraints, which we call *guards*, and a set of timers, which we call *reset timers*. Guards

will be used to model the timing conditions that constrain the execution of transitions. Every reset timer will be reset to 0 whenever this transition is taken.

Besides the transitions that are explicitly represented by the edges and the stuttering transitions, τ, we introduce a special transition called *tick* for representing the elapsing time. Like τ, transition *tick* remains in the source node. For every node n and every active timer on it, we require that the value of every active timer increases whenever *tick* is taken.

Figure 6.2 shows a TPD over $\mathcal{P} = \{x = 0, x \geq 0\}, \mathcal{A} = \{Up, Down\}$, $\Phi_C = \{t \geq 0, t \leq 3\}$. Every node consists of two parts: the first part, above the dashed line, contains the predicates over the system's discrete variables, the second part, below the dashed line, contains all time-constraints in time-invariant that hold on that node. For every edge (n, m) we write $t := 0$ for every timer t in $R(n, m)$.

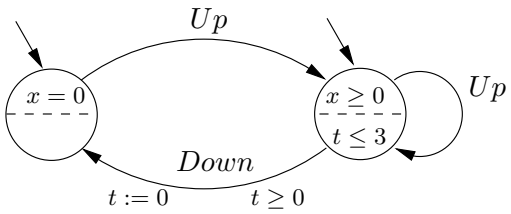

Figure 6.2: An example of TPD.

TPDs can be viewed as an extension of predicate diagrams. In the other direction, we may say that predicate diagrams are restricted TPDs. Particularly, when we eliminate all the components of TPDs that are related to real-time property, then we have predicate diagrams without fairness conditions. We call such a predicate diagram the *untimed version* of a TPD.

Definition 6.3 *(untimed version) Let $T = (N, I, \delta, o, r, g, R)$ be a* TPD *over $\mathcal{P}, \mathcal{A}, C$ and Φ_C. The predicate diagram $G = (N, I, \delta, o, \emptyset)$ over $\mathcal{P}$ and $\mathcal{A}$ is called the untimed version of T.*

We now define *runs* and *traces* through a TPD as the set of behaviors that correspond to runs satisfying the node and edge labels.

Definition 6.4 *(runs, traces) Let $T = (N, I, \delta, o, r, g, R)$ over $\mathcal{P}, \mathcal{A}, C$ and Φ_C as defined. A run of T is ω-sequence $\vartheta = (s_0, n_0, A_0, \Delta_0)(s_1, n_1, A_1, \Delta_1)\ldots$ of quadruples where s_i is a state, $n_i \in N$ is a node, $A_i \in \mathcal{A} \cup \{\tau, tick\}$ is an action and Δ_i is a real number such that all of the following conditions hold:*

1. $n_0 \in I$ *is an initial node.*
2. $s_0[\![c]\!] = 0$ *holds for every* $c \in C$.
3. *For every* $i \in \mathbb{N}$ *hold the following conditions:*
   (a) $s_i[\![n_i \wedge r(n_i)]\!]$.
   (b) *Either* $A_i \in \{\tau, tick\}$ *and* $n_i = n_{i+1}$ *or* $A_i \in \mathcal{A}$ *and* $(n_i, n_{i+1}) \in \delta_{A_i}$.
   (c) *If* $A_i \in \mathcal{A}$ *and* $(t, \prec) \in o(n_i, n_{i+1})$, *then* $s_{i+1}[\![t]\!] \prec s_i[\![t]\!]$.
   (d) *If* $A_i \in \{\tau, tick\}$ *then* $s_{i+1}[\![t]\!] \preceq s_i[\![t]\!]$ *holds whenever* $(t, \prec) \in o(n_i, m)$ *for some* $m \in N$.
   (e) *If* $A_i = \tau$ *then* $\Delta_i = 0$ *and* $s_{i+1}[\![c]\!] = s_i[\![c]\!]$ *holds for every* c *in* C.
   (f) *If* $A_i = tick$ *then* $\Delta_i > 0$ *and* $s_{i+1}[\![c]\!] = s_i[\![c]\!] + \Delta_i$ *holds for every active timer* c *on* n_i *and* $s_{i+1}[\![c]\!] = s_i[\![c]\!]$ *holds for remaining timers.*
   (g) *If* $A_i \in \mathcal{A}$ *then* $\Delta_i = 0$, $s_i[\![g(n_i, n_{i+1})]\!]$ *and* $s_{i+1}[\![c]\!] = 0$ *holds for every* c *in* $R(n_i, n_{i+1})$ *and* $s_{i+1}[\![c]\!] = s_i[\![c]\!]$ *holds for remaining timers .*

We write runs(T) to denote the set of runs of T.

The set $tr(T)$ *of traces through* T *consists of all behaviors* $\sigma = s_0 s_1 \ldots$ *such that there exists a run* $\vartheta = (s_0, n_0, A_0, \Delta_0)(s_1, n_1, A_1, \Delta_1) \ldots$ *of* T *based on the states in* σ.

6.4 Verification

In this section we describe the verification of real-time systems using TPDs.

Assume given a real-time specification *RTSpec* and a property F. We recall that in TLA* formalism, the proof that *RTSpec* satisfies F can be considered as proving the validity of *RTSpec* $\rightarrow F$. Following the approach of the verification of discrete systems using predicate diagrams, we split the proof into two steps: finding a TPD T such that every model of *RTSpec* is a trace through T and then proving that every trace through T is a model of F. The first step is done by considering node and edge labels of predicates on the concrete state space of *RTSpec* and reducing the trace inclusions to a set of first-order verification conditions that concern individual states and transitions. Thus, the first step is done deductively. On the other hand, the second step is done by regarding the node labels related to discrete properties, and probably the auxiliary invariants, as Boolean variables and then encoding

the diagrams a finite labeled transition system. The temporal properties of the system is then can be established by model checking.

6.4.1 Relating specifications and TPDs

To compare a specification and a TPD, we first have to assign meaning to the action names that appear in the diagram. We assume given a function α that assigns an action formula A every action name. Because no confusion is possible, we will leave this assignment implicit, and again write A instead of $\alpha(A)$ when referring to the formula assigned to the name A.

Recalling the general format of real-time specification in Formula 6.8, we put the time constraints explicitly on timed-bounded actions. In the context of TPDs, the situation is different, since we put time constraints on timers and not on actions. To overcome this, we define *bounded-actions* of TPDs. Basically bounded-actions are the same as timed bounded actions, which are actions on which we put time-constraints.

Definition 6.5 *(bounded-action) Let $G = (T, I, \delta, o, r, g, R)$ be a* TPD *over $\mathcal{P}, \mathcal{A}, C$ and Φ_C. An action $A \in \mathcal{A}$ is called a bounded-action if there exists some timer $c \in C$ and two integer numbers d and e such that the following conditions hold:*

- *for every $n \in N$, a predicate of the form $c \leq e$ is in $r(n)$ whenever $n \in En(A)$,*
- *for every $n, m \in N$, a predicate of the form $c \geq d$ is in $g(n, m)$ whenever $(n, m) \in \delta_A$ and*
- *for every $n, m \in N$, c is in $R(n, m)$ whenever $(n, m) \in \delta_A$ or $m \notin En(A)$.*

For a bounded action A, we call c, d and e its corresponding timer, lower-bound and upper-bound, and denote by $clk(A), \ell(A)$ and $\mu(A)$, respectively.

Lemma 6.6 *For some-bounded action A and for every node $n \in N$, $clk(A)$ is an active timer on n if $n \in En(A)$.*

Proof. Let A be a bounded-action and c be a timer such that $c = clk(A)$ and let n be a node such that $n \in En(A)$. Assume that c is not an active timer on n, then there is no node m such that $c \in \psi(g(n, m))$. By assumption $n \in En(A)$, which means that there exists some node m such that $(n, m) \in \delta_A$. Moreover,

by the definition bounded-action, $g(n, m)$ contains a time-constraint of the form $c \geq \ell(A)$, which implies that $c \in \psi(g(n, m))$. Contradiction. ■

We say that a TPD T *conforms* to a specification *RTSpec*, written *RTSpec* $\trianglelefteq T$, if every behavior that satisfies *RTSpec* is a trace through T. The following theorem essentially introduces a set of first-order ("local") verification conditions that are sufficient to establish conformance of a diagram to a real-time system specification in standard form.

Theorem 6.7 *Let* $RTSpec \equiv Init \wedge \Box[Next]_v \wedge RTNow(v) \wedge RT$ *be a real time system specification and* $T = (N, I, \delta, o, r, g, R)$ *be a* TPD *over* $\mathcal{P}, \mathcal{A}, C$ *and* Φ_C *as defined. We say that* T *conforms to RTSpec if the following conditions hold:*

1. $\models Init \rightarrow \bigvee_{n \in I} n.$
2. $\not\approx n \wedge [Next]_v \rightarrow n' \vee \bigvee_{(A,m):(n,m)\in\delta_A} \langle A \rangle_v \wedge m'.$
3. *For all* $n, m \in N$ *and all* $(t, \prec) \in o(n, m)$*:*
   (a) $\not\approx n \wedge m' \wedge \bigvee_{A:(n,m)\in\delta_A} \langle A \rangle_v \rightarrow t' \prec t$ *and*
   (b) $\not\approx n \wedge [Next]_v \wedge n' \rightarrow t' \preceq t.$
4. *For every bounded-action* A*:*
   (a) $\models RTSpec \rightarrow RTBound(A, v, clk(A), \ell(A), \mu(A))$,
   (b) $\models n \rightarrow$ ENABLED $\langle A \rangle_v$ *holds for every node* $n \in En(A)$,
   (c) $\models n \rightarrow \neg$ENABLED $\langle A \rangle_v$ *holds for every node* $n \notin En(A)$,
   (d) $clk(A) \notin act(n)$ *holds for every* $n \in N$ *such that* $n \notin En(A)$,
   (e) $clk(A) \notin \psi(g(n, m))$ *holds for every* $n, m \in N$ *such that* $(n, m) \notin \delta_A$ *and*
   (f) $clk(A) \notin R(n, m)$ *holds for every* $n, m \in N$ *such that* $(n, m) \notin \delta_A$ *and* $m \in En(A)$.
5. $\models \bigwedge_{c \in C} TInit(c) \rightarrow r(n)$ *holds for every* $n \in I$.
6. *For every bounded action* $A \in \mathcal{A}$ *and for every* $n, m \in N$*:*

(a) if $(n, m) \in \delta_A$ *or* $m \notin En(A)$ *then*

$$\approx r(n) \wedge clk(A) \geq \ell(A) \wedge clk(A)' = 0 \wedge \bigwedge_{A_1 : clk(A_1) \in act(m)} clk(A_1)' \leq \mu(A_1) \rightarrow r(m)',$$

(b) otherwise,

$$\approx r(n) \wedge \bigwedge_{A_1 : clk(A_1) \in act(n)} clk(A_1)' \geq clk(A_1) \wedge clk(A_1)' \leq \mu(A_1) \rightarrow r(n)'.$$

Proof. Assume that *RTSpec* and T are such that all the conditions hold, and that for every timer c in C, there exists some bounded-action A such that $c = clk(A)$. Assume $\sigma = s_0 s_1 \ldots$ is a behavior that satisfies *RTSpec*. We want to show that $\sigma \in tr(T)$ by constructing the corresponding run through T for σ.

We inductively define a sequence $n_0, n_1, \ldots$ of nodes $n_i \in N$ and a sequence $\mathcal{A}_0 \mathcal{A}_1 \ldots$ of sets of actions $\emptyset \neq \mathcal{A}_i \subseteq \mathcal{A} \cup \{\tau, tick\}$ such that for all $i \in \mathbb{N}$ the following conditions hold:

(i) $n_0 \in I$
(ii) $s_i[\![n_i]\!]$
(iii) $\mathcal{A}_i = \{A \in \mathcal{A} : (n_i, n_{i+1}) \in \delta_A, s_i[\![\langle A \rangle_v]\!] s_{i+1}$ and $s_i[\![now]\!] = s_{i+1}[\![now]\!]\}$
$\cup \{tick : n_i = n_{i+1}$ and $s_i[\![now]\!] \neq s_{i+1}[\![now]\!]\}$
$\cup \{\tau : n_i = n_{i+1}$ and $s_i[\![now]\!] = s_{i+1}[\![now]\!]\}$
(iv) $s_{i+1}[\![t]\!] \prec s_i[\![t]\!]$ whenever $(t, \prec) \in o(n_i, n_{i+1})$ and
(v) $s_{i+1}[\![t]\!] \preceq s_i[\![t]\!]$ whenever $n_{i+1} = n_i$ and $(t, \prec) \in o(n_i, m)$ for some $m \in N$.

For the induction base, we choose some node $n_0 \in I$ such that $s_0[\![n_0]\!]$ holds. The existence of some such node is ensured by condition 1 since $s_0[\![Init]\!]$ holds by assumption.

For the induction step, we assume that $n_0, n_1 \ldots n_i$ and $\mathcal{A}_0, \mathcal{A}_1 \ldots \mathcal{A}_{i-1}$ have already been defined such that conditions (i)-(ii) hold for all $j \leq i$ and conditions (iii)-(v) holds for all $j \leq i - 1$. In particular, we have $s_i[\![n_i]\!]$. Moreover, the assumption that $\sigma \models$ *RTSpec* implies that $s_i[\![[Next]_v]\!] s_{i+1}$, and condition 2 in Theorem 6.7 ensures that either there exist some action $A \in \mathcal{A}$ and some $n \in N$ such that $(n_i, n) \in \delta_A$ and $s_i[\![\langle A \rangle_v \wedge n']\!] s_{i+1}$ holds, or $s_{i+1}[\![n_i]\!]$. In the first case, choose some such node n as n_{i+1}; in the second case, choose $n_{i+1} = n_i$. In either case, define $\mathcal{A}_i$ as described in condition (iii). These choices imply that $s_{i+1}[\![n_{i+1}]\!]$ and that $\mathcal{A}_i \neq \emptyset$. Conditions (iv) and (v) now follow from the choices of n_{i+1} and $\mathcal{A}_i$ with the help of conditions 3a and 3b in Theorem 6.7.

We have shown that conditions 1, 3b, 3c and 3d in the Definition 6.4 hold. The assumption that $\sigma \models$ *RTSpec*, the condition 4a in Theorem 6.7 and the assumption for every timer c there exists some bounded-action A such that $c = clk(A)$, imply

that $s_0[\![\mathit{TInit}(c)]\!]$ holds for every $c \in C$. This also implies that for every $c \in C$, $s_0[\![c]\!] = 0$ holds. Thus the condition 2 in Definition 6.4 holds.

To prove that the conditions 3e-3g in Definition 6.4 hold, we define a sequence $\Delta_0\Delta_1\ldots$ of integer numbers such that $\Delta_i = s_{i+1}[\![\mathit{now}]\!] - s_i[\![\mathit{now}]\!]$ and pick a sequence of actions $A_0A_1\ldots$, such that $A_i \in \mathcal{A}_i$ holds for every $i \in \mathbb{N}$. Choose some $i \in \mathbb{N}$ and consider the cases of $A_i = \tau$, $A_i = \mathit{tick}$ and $A_i \in \mathcal{A}$.

If $A_i = \tau$ then we have $n_i = n_{i+1}$ and $s_i[\![\mathit{now}]\!] = s_{i+1}[\![\mathit{now}]\!]$. By the construction of $\Delta_0\Delta_1\ldots$, this also implies that $\Delta_i = 0$. Choose some bounded-action $A \in \mathcal{A}$ and consider the following cases:

- Assume $n_i \in \mathit{En}(A)$. Since $n_{i+1} = n_i$, we have $n_{i+1} \in \mathit{En}(A)$. Condition 4a implies that $s_i[\![\mathit{TNext}(A, v, \mathit{clk}(A))]\!]s_{i+1}$ holds, since by assumption $\sigma \models \mathit{RTSpec}$. Condition 4b ensures that $s_{i+1}[\![\text{ENABLED } \langle A\rangle_v]\!]$ holds. Moreover, since $s_i[\![\mathit{now}]\!] = s_{i+1}[\![\mathit{now}]\!]$, we have $s_{i+1}[\![\mathit{clk}(A)]\!] = s_i[\![\mathit{clk}(A)]\!]$.

- Assume $n_i \notin \mathit{En}(A)$. Then we have $n_{i+1} \notin \mathit{En}(A)$, which by condition 4c, implies that $s_{i+1}[\![\neg\text{ENABLED } \langle A\rangle_v]\!]$ or $s_i[\![\neg\text{ENABLED } \langle A\rangle'_v]\!]$. By assumption that $\sigma \models \mathit{RTSpec}$ and by condition 4a, $s_i[\![\mathit{TNext}(A, v, \mathit{clk}(A))]\!]s_{i+1}$ holds, which implies that $s_{i+1}[\![\mathit{clk}(A)]\!] = 0$. Again, we have two cases to consider. First, assume that $i = 0$. We have shown that $s_0[\![\mathit{clk}(A)]\!] = 0$. Since $s_{i+1}[\![\neg\text{ENABLED } \langle A\rangle_v]\!]$, we have $s_{i+1}[\![\mathit{clk}(A)]\!] = 0$. Second, assume that $i > 0$. By condition 4c, the condition $n_i \notin \mathit{En}(A)$ implies that $s_i[\![\neg\text{ENABLED } \langle A\rangle_v]\!]$, or we can say, $s_{i-1}[\![\neg(\text{ENABLED } \langle A\rangle_v)']\!]$ which implies that $s_i[\![\mathit{clk}(A)]\!] = 0$. Thus, for both cases, we have $s_i[\![\mathit{clk}(A)]\!] = s_{i+1}[\![\mathit{clk}(A)]\!] = 0$.

For every bounded-action A, we have shown that $s_i[\![\mathit{clk}(A)]\!] = s_{i+1}[\![\mathit{clk}(A)]\!]$. By the assumption that for every $c \in C$ there exists some bounded action A such that $\mathit{clk}(A) = c$, the condition 3e in Definition 6.4 holds.

In the case of $A_i = \mathit{tick}$, by the construction of $\mathcal{A}_0\mathcal{A}_1\ldots$, we have $s_i[\![\mathit{now}]\!] \neq s_{i+1}[\![\mathit{now}]\!]$. The assumption that $\sigma \models \mathit{RTSpec}$ implies that $s_{i+1}[\![\mathit{now}]\!] > s_i[\![\mathit{now}]\!]$, which also implies that $\Delta_i > 0$. Do the similar proof as above, we can prove that for every bounded-action A such that $n_i \in \mathit{En}(A)$, $s_{i+1}[\![\mathit{clk}(A)]\!] > s_i[\![\mathit{clk}(A)]\!]$, and for every bounded action A such that $n_i \notin \mathit{En}(A)$, $s_{i+1}[\![\mathit{clk}(A)]\!] = s_i[\![\mathit{clk}(A)]\!]$. By Lemma 6.6 and by the assumption that for every $c \in C$ there exists some bounded action A such that $\mathit{clk}(A) = c$, we can conclude that the condition 3f in Definition 6.4 holds.

In the case of $A_i \in \mathcal{A}$, by the construction of $\mathcal{A}_0\mathcal{A}_1\ldots$ and $\Delta_0\Delta_1\ldots$, we have $s_i[\![\mathit{now}]\!] = s_{i+1}[\![\mathit{now}]\!]$ and $\Delta_i = 0$. Choose some bounded action $A \in \mathcal{A}$ and consider the following cases:

- Assume that $A \in A_i$. This implies that $(n_i, n_{i+1}) \in \mathit{En}(A)$. By the definition of bounded-action, we have $\mathit{clk}(A) \in R(n_i, n_{i+1})$. By the construction

of $\mathcal{A}_0\mathcal{A}_1\ldots$, $s_i[\![\langle A\rangle_v]\!]s_{i+1}$ holds. The assumption that σ satisfies *RTSpec* and condition 4a imply that $s_i[\![TNext(A, v, clk(A))]\!]s_{i+1}$ holds. Moreover, since $s_i[\![\langle A\rangle_v]\!]s_{i+1}$, we have $s_{i+1}[\![clk(A)]\!] = 0$.

- Assume that $A \notin \mathcal{A}_i$. If $n_{i+1} \in En(A)$ then by the definition of bounded-action, $clk(A) \in R(n_i, n_{i+1})$, and we have shown that $s_{i+1}[\![clk(A)]\!] = 0$. If $n_{i+1} \notin En(A)$ then by condition 4e we have $clk(A) \notin R(n_i, n_{i+1})$ and by construction of $\mathcal{A}_0\mathcal{A}_1\ldots$, $s_i[\![\neg\langle A\rangle_v]\!]s_{i+1}$ holds. With the same argument as before, the condition $s_i[\![TNext(A, v, clk(A))]\!]s_{i+1}$ holds, and as consequence we have $s_{i+1}[\![clk(A)]\!] = s_i[\![clk(A)]\!]$, since $s_{i+1}[\![now]\!] = s_i[\![now]\!]$.

  We have shown that for both cases, the conditions that are related to reset timers which are required in condition 4g in Definition 6.4 hold.

The definition of bounded-actions and the condition 4f in Theorem 6.7 ensure that $g(n_i, n_{i+1})$ contains only the time-constraints over timers which are the corresponding timers of some bounded-actions A such that $(n_i, n_{i+1}) \in \delta_A$. In particular, $g(n_i, n_{i+1})$ contains time-constraints of the form $clk(A) \geq \ell(A)$. The assumption that $\sigma \models RTSpec$ and condition 4a imply that for every bounded-action $s_i[\![A \rightarrow clk(A) \geq \ell(A)]\!]$. The condition 3g in Definition 6.4 holds.

It remains to prove that $s_i[\![n_i \wedge r(n_i)]\!]$ holds for every $i \in \mathbb{N}$. We already know that $s_i[\![n_i]\!]$ holds for every $i \in \mathbb{N}$. We have shown that $s_0[\![TInit(c)]\!]$ holds for every c in C. By condition 5, this implies that $s_0[\![r(n_0)]\!]$ holds. Consider again the sequence of $A_0A_1\ldots$ defined above, choose some $j \in \mathbb{N}$ and assume that $s_i[\![r(n_i)]\!]$ holds for every $i \leq j$. If $A_j \in \mathcal{A}$ then by the definition of bounded-actions $clk(A) \in R(n_j, n_{j+1})$ and $r(n_{j+1})$ contains time-constraints of the form $clk(A) \leq \mu(A)$ for every bounded-action A such that $n_{j+1} \in En(A)$. Condition 6a ensures that $s_{j+1}[\![r(n_{j+1})]\!]$ holds. If $A_j = \tau$, we have shown that for every bounded-action A, $s_{j+1}[\![clk(A)]\!] = s_j[\![clk(A)]\!]$ and if $A_j = tick$, we have shown that for every bounded-action A, either $s_{j+1}[\![clk(A)]\!] > s_j[\![clk(A)]\!]$ whenever $n_j \in En(A)$ or $s_{j+1}[\![clk(A)]\!] = s_j[\![clk(A)]\!]$ whenever $n_j \notin En(A)$. For both cases, condition 6b ensures that $s_{j+1}[\![r(n_{j+1})]\!]$ holds. Thus, the condition 3a in Definition 6.4 holds which completes the proof. ■

The first three conditions in Theorem 6.7 are inherited from Theorem 5.1, the conformance theorem of predicate diagrams. Those conditions are related to the discrete properties of the system. Conditions 4a-4f are related to the bounded-actions in the diagram and the two last conditions are related to the time-invariants in the diagrams.

Condition 5 ensures that the time-invariant of every initial node is implied by the initial condition of every timer. Condition 6 guarantees that the time-invariants always agree with the changes of the concrete values of the timers

whenever a transition is taken. In particular, the condition 6a requires that for every time-bounded action A, its corresponding timer should be reset to 0 whenever A is taken or it is not enabled at the next state and the value of every active timer on the next state is less than or equal to its corresponding upper-bound; and for the other cases condition 6b requires that the value of each action timer on the next state should be greater than or equal to its current value and less than or equal to its upper bound.

Theorem 6.7 can be used to show that the TPD of Figure 6.2 conforms to the specification LOOP in Figure 6.1. For example, we have:

- $\mathit{Init} \rightarrow x = 0 \vee x > 0$.
- $x = 0 \wedge [\mathit{Next}]_x \rightarrow x' = 0 \vee \langle \mathit{Up} \rangle_x \wedge x' > 0$.
- $x > 0 \wedge [\mathit{Next}]_x \rightarrow x' > 0 \vee \langle \mathit{Down} \rangle_x \wedge x' = 0$.
- $\mathit{Loop} \rightarrow \mathit{RTBound}(\mathit{Down}, x, t, 0, 3)$.
- $t \leq 3 \wedge t \geq 0 \wedge t' = 0 \wedge \mathbf{true} \rightarrow t' \leq \infty$.
- $t \leq 3 \wedge t' \geq t \wedge t' \leq 3 \rightarrow t' \leq 3$.

6.4.2 Model checking TPDs

For the proof that all traces through a TPD satisfy some property F, we view the TPD as a finite transition system that is amenable to model checking.

We propose two approaches for model checking TPDs. First, if the quantitative aspect of times doesn't come into account, it is enough to model check their untimed versions which are predicate diagrams. We have discussed this issue in Section 5.4.2.

However, whenever we have to consider the quantitative aspect of times for proving the properties we want to verify, we will use some existing real-time system model-checker for verifying TPDs. For example, we can use Kronos, which is a software tool built with the aim of assisting designers of real-time systems to verify whether their designs meet the specified requirements [110]. To do that, we first translate our diagrams into the input of Kronos, which are timed automata (see Section 4.4), with some additional information. A timed automaton Γ is now given by a tuple $(Q, Q_0, \mathcal{X}, \mathcal{I}, \Lambda, P)$. Except P, all components of Γ is as defined in Definition 4.22. P is a function associates with each location a set of atomic propositions.

Given a TPD T over $\mathcal{P}, \mathcal{A}, C$ and Φ_C. The translation from T to some timed-automata can be done by using this following construction.

Construction 6.8 *Let $T = (N, I, \delta, o, r, g, R)$ be a* TPD *over $\mathcal{P}, \mathcal{A}, C$ and Φ_C. Let $\mathcal{A}^n$ be a set containing action names and $\kappa : \{1..|N|\} \to N$ be an injective function which associates every node in N with some natural number.*

One can construct the corresponding timed automaton $\Gamma = (Q, Q_0, \mathcal{X}, \mathcal{I}, \Lambda, P)$ as follows:

- $Q = \{q_1, \ldots, q_{|N|}\}$.
- $Q_0 = \{q_i : \kappa(i) \in I\}$.
- $\mathcal{X} = C$.
- *For every $i \in 1..|N|$, $P(q_i) = \kappa(i) \wedge r(\kappa(i))$ and $\mathcal{I}(q_i) = r(\kappa(i))$.*
- *For every $(n, m) \in \delta$ and for every $A \in \mathcal{A}$, there exists some tuple $(q_i, A, \lambda, \theta, q_j)$ in Λ such that $\kappa(i) = n$, $\kappa(j) = m$, $\lambda = R(n, m)$ and $\theta = g(n, m)$.*
- *For every $n \in N$, there exists some tuple of the form $(q_i, a, \lambda, \theta, q_i)$ in Λ such that $\kappa(i) = n$, $a = \{tick, \tau\}$, $\lambda = R(n, n)$ and $\theta = g(n, n)$.*

Theorem 6.9 *Let $T = (N, I, \delta, o, r, g, R)$ be a* TPD *over $\mathcal{P}, \mathcal{A}, C$ and Φ_C as defined and let $\Gamma = (Q, Q_0, \mathcal{X}, \mathcal{I}, \Lambda, P)$ be the resulted automaton from Construction 6.8 over T. For every run through T, $\rho = (s_0, n_0, A_0, \Delta_0)(s_1, n_1, A_1, \Delta_1)\ldots$, there exists a run of Γ*

$$\phi = (q_0, \nu_0) \overset{w_0, \varphi_0}{\longrightarrow} (q_1, \nu_1) \overset{w_1, \varphi_1}{\longrightarrow} \ldots$$

such that $\kappa(i) = n_i$ and $s_i[\![C]\!] = \nu_i(\mathcal{X})$ holds for every $i \in \mathbb{N}$.

Proof. Let $\rho = (s_0, n_0, A_0, \Delta_0)(s_1, n_1, A_1, \Delta_1)\ldots$ be a run through T. We define two sequences $w = w_0 w_1 \ldots$ and $\varphi = \varphi_0 \varphi_1 \ldots$ such that $w_i = A_i$ and $\varphi_i = \Delta_{i+1} - \Delta_i$ holds for every $i \in \mathbb{N}$. Construction 6.8 ensures that we can construct a run of Γ as stated in Theorem 6.9. ■

6.5 An example: Fischer's protocol

As illustration we take the FISCHER's mutual exclusion protocol which is a well-known and well-studied by researchers in the context of real-time verification. We take the simplified version which only two processes in the protocol.

module FISCHER

Specification

$$
\begin{array}{rl}
\mathit{Init} \equiv & x = 0 \land pc_1 = 0 \land pc_2 = 0 \\
\mathit{Try}_1 \equiv & x = 0 \land pc_1 = 0 \land pc_1' = 1 \land x' = x \land pc_2' = pc_2 \\
\mathit{Set}_1 \equiv & pc_1 = 1 \land x' = 1 \land pc_1' = 2 \land pc_2' = pc_2 \\
\mathit{Enter}_1 \equiv & \land\ pc_1 = 2 \land x' = x \land pc_2' = pc_2 \\
& \land \lor\ x = 1 \land pc_1' = 3 \\
& \quad \lor\ x \neq 1 \land pc_1' = 0 \\
\mathit{Exit}_1 \equiv & pc_1 = 3 \land pc_1' = 0 \land x' = 0 \land pc_2' = pc_2 \\
\mathit{Try}_2 \equiv & x = 0 \land pc_2 = 0 \land pc_2' = 1 \land x' = x \land pc_1' = pc_1 \\
\mathit{Set}_2 \equiv & pc_2 = 1 \land x' = 2 \land pc_2' = 2 \land pc_1' = pc_1 \\
\mathit{Enter}_2 \equiv & \land\ pc_2 = 2 \land x' = x \land pc_1' = pc_1 \\
& \land \lor\ x = 2 \land pc_2' = 3 \\
& \quad \lor\ x \neq 2 \land pc_2' = 0 \\
\mathit{Exit}_2 \equiv & pc_2 = 3 \land pc_2' = 0 \land x' = 0 \land pc_1' = pc_1 \\
\mathit{Next} \equiv & \lor\ \mathit{Try}_1 \lor \mathit{Set}_1 \lor \mathit{Enter}_1 \lor \mathit{Exit}_1 \\
& \lor\ \mathit{Try}_2 \lor \mathit{Set}_2 \lor \mathit{Enter}_2 \lor \mathit{Exit}_2 \\
v \equiv & \langle x, pc_1, pc_2 \rangle \\
\mathit{Fischer} \equiv & \mathit{Init} \land \Box[\mathit{Next}]_v \land \mathit{RTNow}(v) \\
& \qquad \land \mathit{RTBound}(\mathit{Set}_1, v, s_1, 0, D) \\
& \qquad \land \mathit{RTBound}(\mathit{Enter}_1, v, t_1, E, \infty) \\
& \qquad \land \mathit{RTBound}(\mathit{Set}_2, v, s_2, 0, D) \\
& \qquad \land \mathit{RTBound}(\mathit{Enter}_2, v, t_2, E, \infty)
\end{array}
$$

Theorem

1. $\mathit{Fischer} \rightarrow \Box\neg(pc_1 = 3 \land pc_2 = 3)$

Figure 6.3: FISCHER's protocol.

The system is composed by a set of 2 timed processes, P_1 and P_2 plus a shared variable x. Each process P_i behaves as follow: after remaining idle for some time, it checks whether the common resource is free (test $x = 0$) and if so, before D time units sets x to i. Then it waits at least for E time units and, making sure that x is still equal to i, enters the critical section. If x is not equal to i (meaning that some other process has requested access) then process i has to retry later.

The module contains the specification of the protocol and some properties to be verified is given in Figure 6.3.

For every process P_i we associate a variable pc_i which represents its control state. The value of pc_i is equal to one of the integer $0, 1, 2$ or 3. The initial predicate *Init* asserts that pc_i is equal to 0 for each process i, so the processes start with control at statement 0.

We need to verify that, with suitable conditions on D and E there will never be more than one process in its critical section. This property can be

expressed as a formula:

$$Fischer \rightarrow \Box\neg(pc_1 = 3 \wedge pc_2 = 3). \tag{6.9}$$

Figure 6.4 depicts the untimed version of TPD for Fischer's protocol. We will generate a suitable TPD of this predicate diagram. Since in the specification $Set_1, Enter_1, Set_2$ and $Enter_2$ appear as time-bounded actions, we intuitively treat those actions as bounded-actions in the TPD. For each bounded-action we set the corresponding clock, lower bound and upper bound as follows:

- $clk(Set_1) = s_1, \ell(Set_1) = 0, \mu(Set_1) = D$,
- $clk(Enter_1) = t_1, \ell(Enter_1) = E, \mu(Enter_1) = \infty$,
- $clk(Set_2) = s_2, \ell(Set_2) = 0, \mu(Set_2) = D$, and
- $clk(Enter_2) = t_2, \ell(Enter_2) = E, \mu(Enter_2) = \infty$.

Based on the information about the lower and upper bound of every timed-bounded action, we can add the time-invariants and guards for the suitable nodes. For example, at nodes where Set_1 can be taken, we add time invariant $s_1 \leq D$ and on edges along where Set_1 can be taken we put a guard $s_1 \geq 0$ and reset timer $s_1 := 0$.

Figure 6.5 shows the TPD for the FISCHER problem where $clk(Set_1) = s_1, clk(Set_2) = s_2, clk(Enter_1) = t_1$ and $clk(Enter_2) = t_2$. Notice that due to the limited space, for every edge (n, m) and for every timer c, we only label (n, m) with $c := 0$ whenever c is active in n. The diagram represents an approximation of the FISCHER specification. It conforms to the specification, because every behavior of the FISCHER specification is a trace through the diagram. We can use the conformance theorem, Theorem 6.7, for proving this conformance.

However, it is too weak: it cannot be used to prove the mutual exclusion property (Formula 6.9).

Our first try is to extend the diagram in Figure 6.5 by adding some more time-constraints to every node. Suppose as additional time-invariants for every node we take the initial values of every timer, i.e $s_1 = 0, s_2 = 0, t_1 = 0$ and $t_2 = 0$; and also two predicates asserting comparison between s_1 and t_2 and comparison between s_2 and t_1, i.e. $s_1 = t_2$ and $s_2 = t_1$. Starting with the initial node we traverse the diagram in order to determine whether those time-invariants still hold on each node or not. We do this by considering the

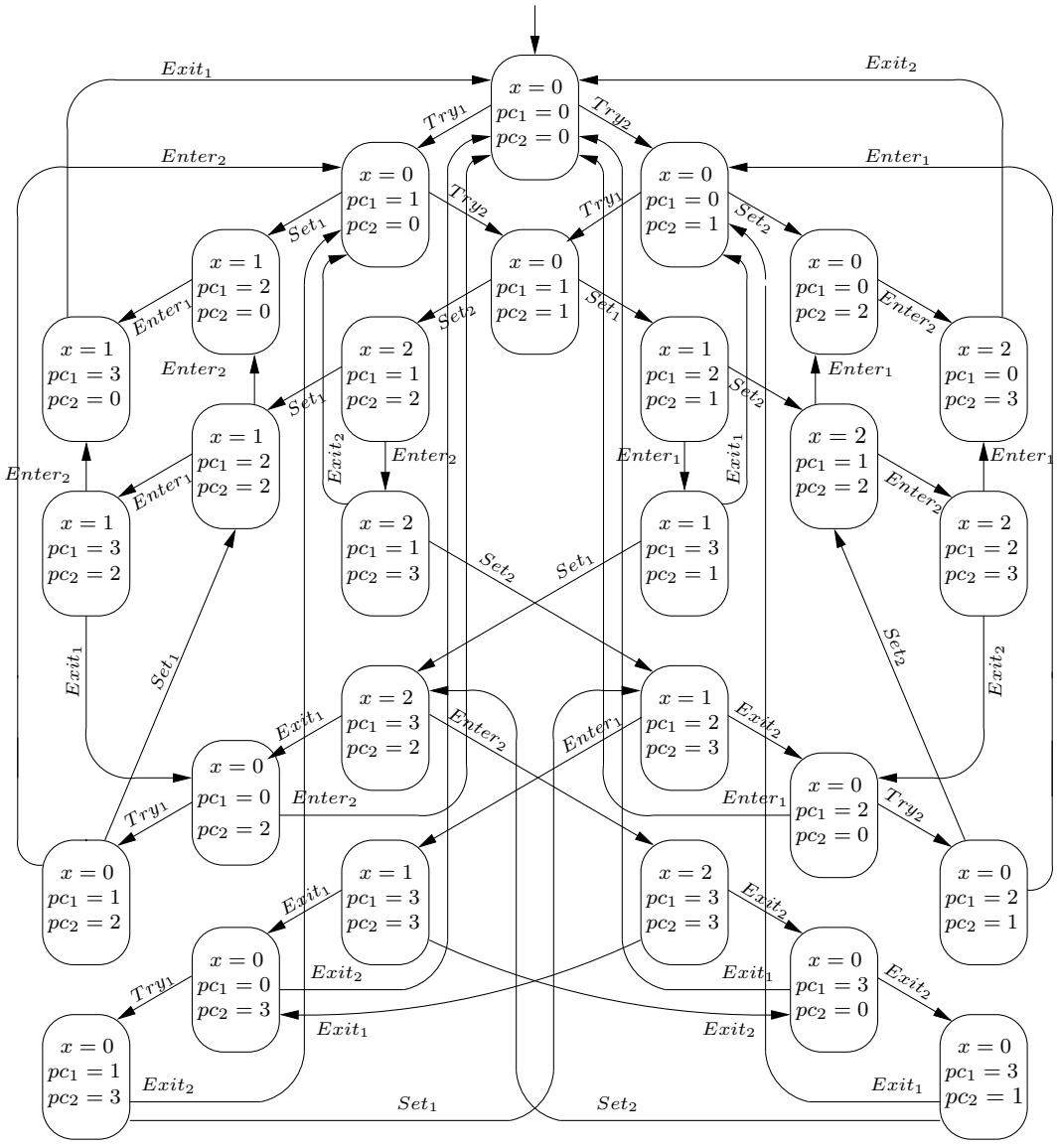

Figure 6.4: Predicate diagram for FISCHER's protocol.

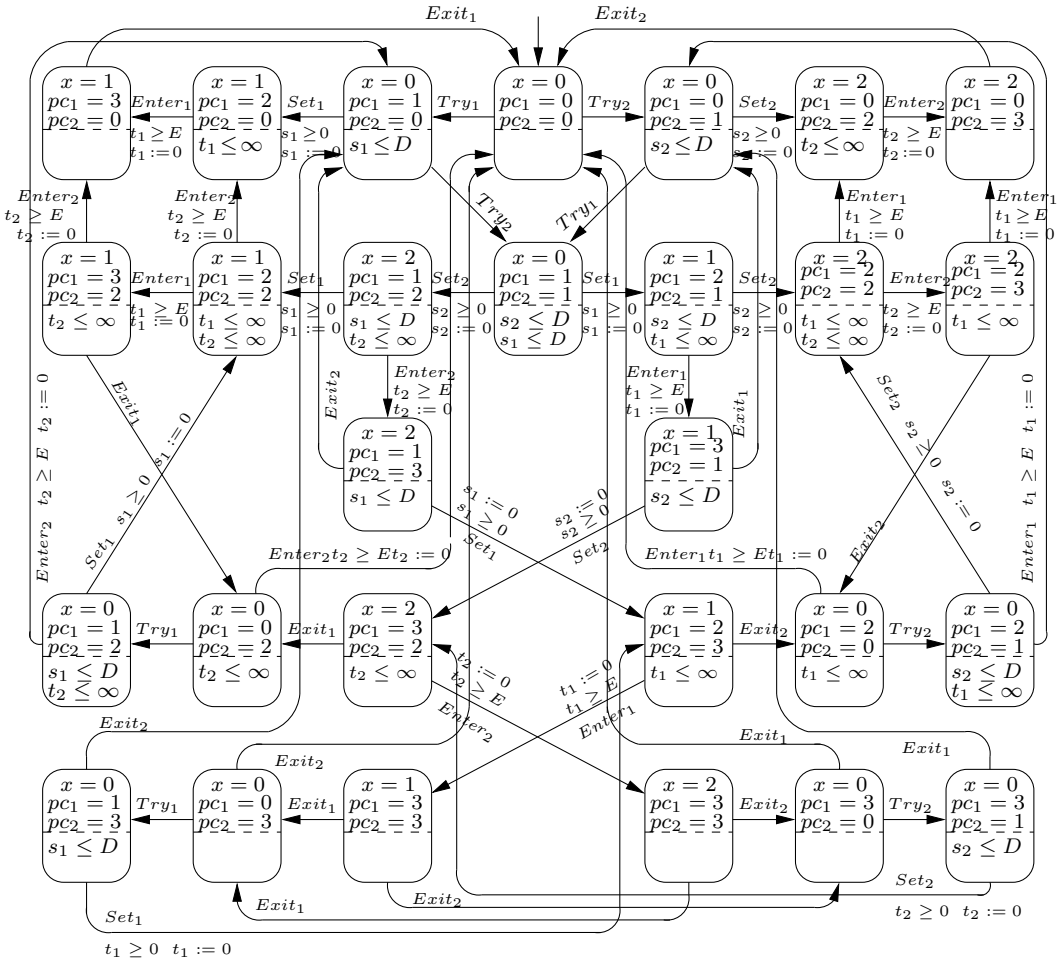

Figure 6.5: First TPD for FISCHER's protocol.

bounded-actions that can be taken on every node. For example, if Set_1 is enabled then $s_1 = 0$ is changed to $s_1 \geq 0$. The predicates $s_1 = t_2$ and $s_2 = t_1$ might be changed accordingly.

The result is shown in Figure 6.6 (the numbering on some nodes will be used later). The resulted diagram, again, represents an approximation of the FISCHER specification. However, we still cannot prove the mutual exclusion property.

We can strengthen the second diagram, by using assumptions on D and E. We consider two cases: $D < E$ and $D \geq E$. Based on these assumptions, we refine our diagram in Figure 6.6 by eliminating transitions that do not satisfy those conditions. The elimination process can be done using SIMPLIFY algorithm in Figure 6.7. The algorithm works in DFS style. For every initial node n, we run the algorithm with TPD T, some node n and some set of edge δ_1 as inputs. Starting with initial nodes, for every edge leading from this node, the algorithm checks whether it satisfies the required condition or not. If so, then the edge is stored and the algorithm continues to check the next edges. For example, consider node 1 and node 2 in Figure 6.6. The corresponding edge does not satisfy either condition 6a, since it is never the case that $s_1 \geq t_2 \wedge s_1 \leq D \wedge t_2 \geq E$ is true while $D < E$, or condition 6b, since in this case node 1 and node 2 are different nodes. The situation is the similar with node 3 and node 4. Running this algorithm over our second diagram and assumption $D < E$, we have the third diagram shown in Figure 6.8.

For the assumption $D \geq E$, the resulted diagram has the same structure with the one in Figure 6.6. As conclusion, the protocol satisfies mutual exclusion property if $D < E$.

Since we don't consider the quantitative aspect of time, we may work with the untimed version of TPD shown in Figure 6.8 and then model-check the resulted diagram as explained in Section 5.4.2.

6.6 Discussions and related work

We have presented a method for the verification of real-time systems. A real-time program can be written as the conjunction of its untimed version, expressed in a standard way as a TLA* formula, and its timing assumptions, expressed in terms of a few standard parameterized formulas. The separation between specification of untiming and timing properties makes real-time specification easier to write and understand.

We have defined a variant of predicate diagrams presented in the previous

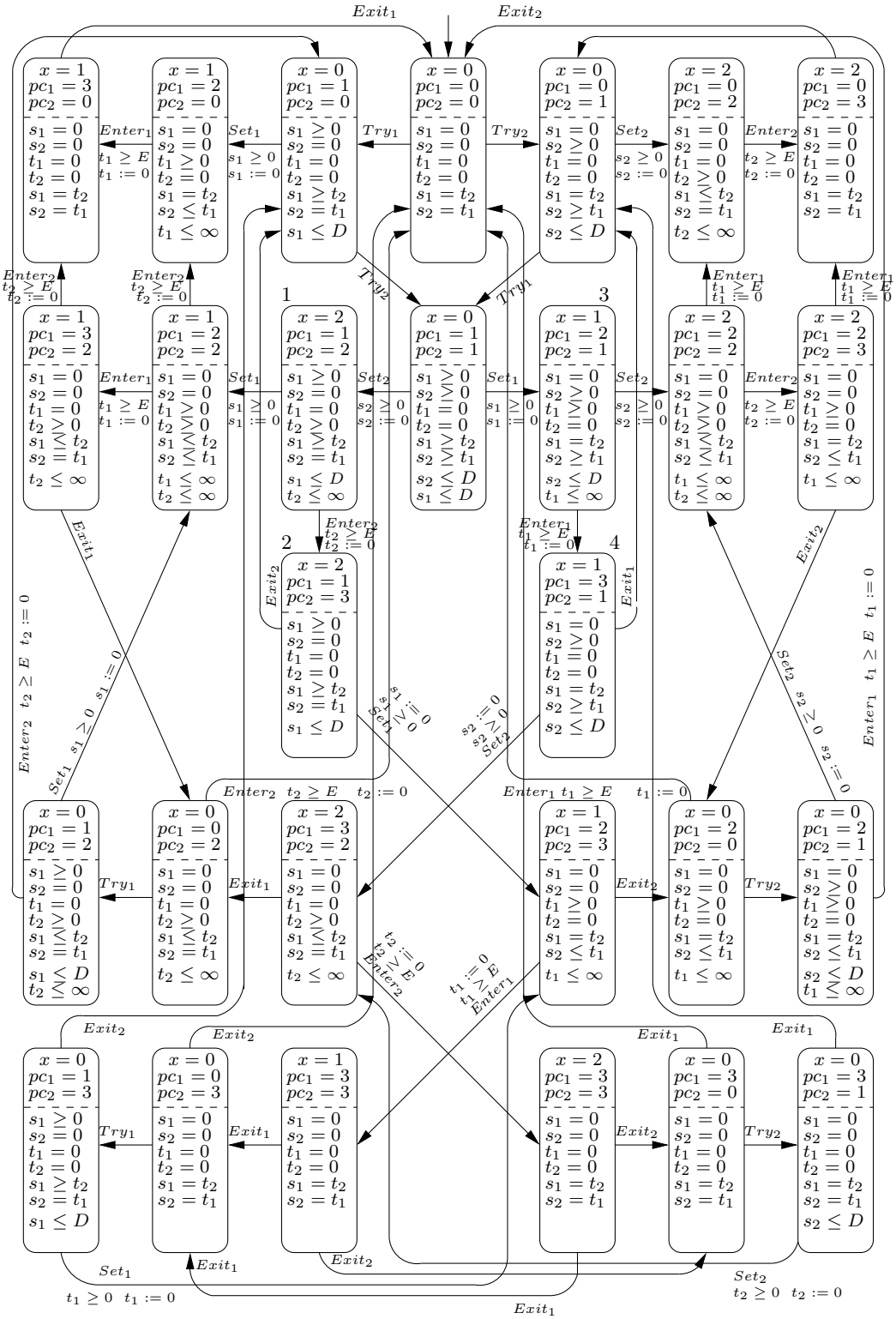

Figure 6.6: Second TPD for FISCHER's protocol.

algorithm SIMPLIFY

```
if n ∉ visited then
    visited = visited ∪ {n}
    for every (n, m) ∈ δ
        if either condition 6a or 6b in Theorem 6.7 is satisfied then
            δ1 ≡ δ1 ∪ {(n, m)}
            SIMPLIFY(T, m, δ1, visited)
        endif
    endfor
endif
```

Figure 6.7: SIMPLIFY algorithm.

chapter, which we call timed predicate diagrams or TPDs. Basically, TPDs are predicate diagrams which dedicated to handle real-time systems. The properties of timed predicate diagrams, except the ones that related to real-time condition, are inherited from predicate diagrams.

The verification process of real-time systems using TPDs is quite similar to the one of discrete systems. Like predicate diagrams, TPDs also integrate the deductive and algorithmic verification techniques. For proving that a TPD conforms to a real-time specification, we can use Theorem 6.7. There are two possible ways in model-checking TPDs. If the quantitative aspect of time comes into account then we can use some existing real-time systems model-checkers, such as KRONOS [110] or UPPAAL [15], for model-checking TPDs. To do that we need to translate our diagram into the input languages of the model-checkers, which are timed automata. Since the structure of TPDs are quite similar to timed automata, the translation can be done straightforward using the setting described in section 6.4.2. If we don't consider the quantitative aspect of time, we may work with the untimed version of TPD which are ordinary predicate diagrams and use the standard model-checker such as SPIN to model-check the diagrams. We have illustrated the latter case when we proved the mutual exclusion property of the FISCHER's protocol.

Many models for reasoning real-time systems have been proposed. The approach to real-time presented in MANNA et al. in [55] and [77] is based on the computational model of *Timed Transition Systems* (TTS) in which time itself is not explicitly represented but it is reflected in a time stamp affixed to each state in a computation of a TTS. In [59], KESTEN et al. introduced a computation model for real-time systems called *Clocked Transition Systems*

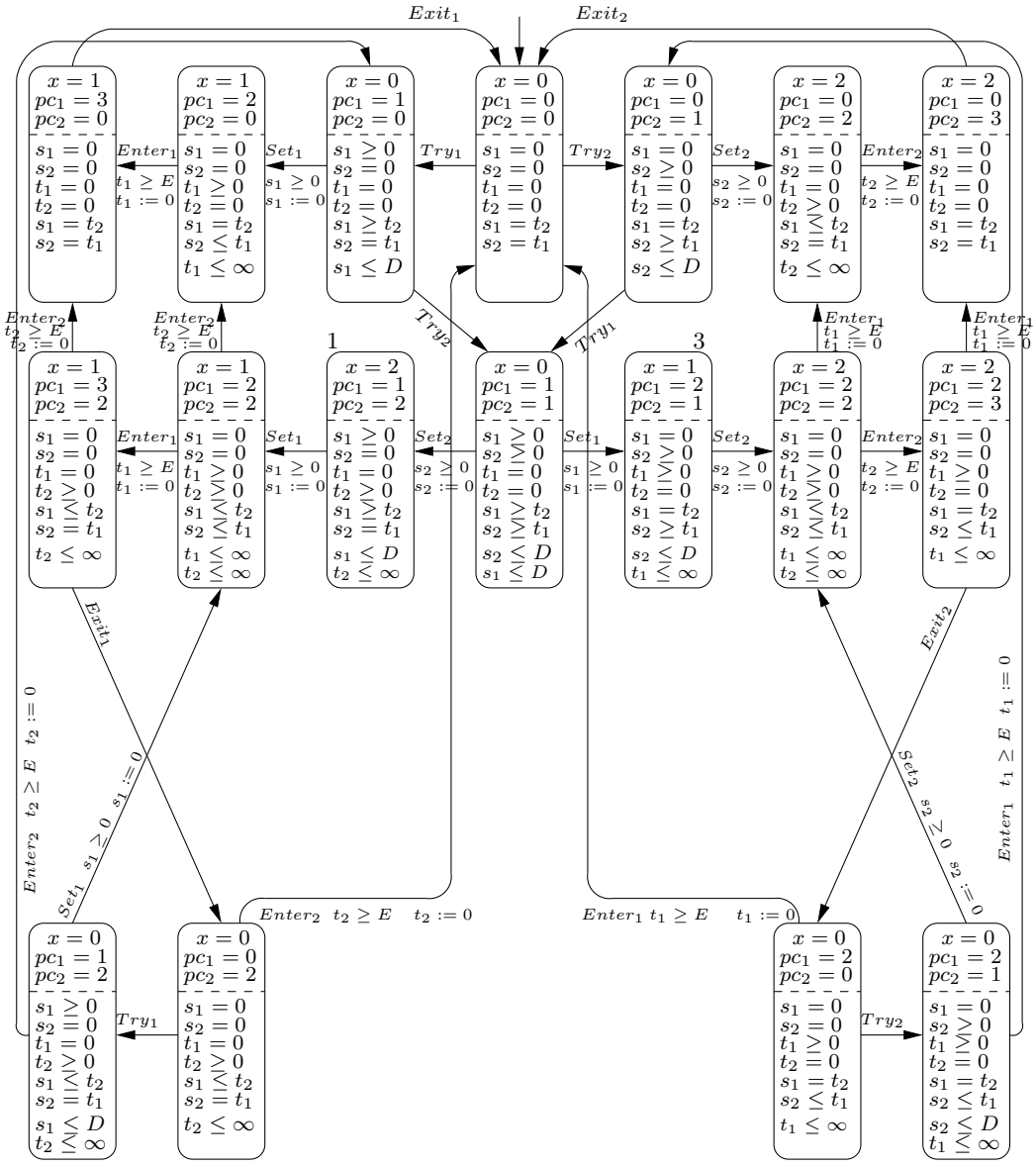

Figure 6.8: TPD for FISCHER's protocol with assumption $D < E$.

(CTS), which is a development of TTS. This model represents time by a set of system variables called clocks (timers) which increase uniformly whenever time progresses, but can be set to arbitrary values by system (program) transitions.

Modelling timer/clocks as just another kind of system variables has also been proposed by ABADI & LAMPORT, which is followed in this work. Such models bring two benefits: it leads to a more natural style of specification, instead of introducing special new constructs (e.g. bounded temporal operators proposed in metric temporal logic (MTL) [64, 63, 62] or the age function proposed in [77]), and we can reuse many of the methods and tools developed for verifying untimed reactive systems.

ALUR&DILL in [9] proposed an automata based approach for reasoning real-time systems. They introduced *timed automata*, which are an extension of ω-automata (see Chapter 3). Timed automata are automata equipped with a set of variables, called clocks, that measure the time elapsed on locations.

We refer the reader to [10, 89] and the survey in [11], for additional logics, models and approaches to the verification of real-time systems.

The use of diagrams in verification of real-time systems can be found, for example, in [59]. In their approach, they use a special rule for proving a class of property, such as invariant and response properties. Every rule is associated with verification diagram. In her thesis, SIPMA presented two specialized classes of diagrams for real-time systems: nonZenoness and receptiveness diagrams. NonZenoness diagrams represent a proof that a real-time is time-divergent, that is, all behavior prefixes of the system can be extended into behaviors in which time grows beyond any bound; whereas receptiveness diagrams prove a related property of real-time systems that implies time divergence and is preserved by parallel composition.

The generation of the TPDs for the FISCHER's problem is done manually. In Chapter 8, we will study some techniques for generating diagrams, or invariants, (semi-)automatically. BOUAJJANNI et al., see [104], have successfully generated the invariants for the FISCHER's protocol automatically using TReX, a tool for reachability analysis of complex systems.

Chapter 7

Parameterized systems

7.1 Overview

Parameterized systems have become a very important subject of research in the area of computer-aided verification. A typical parameterized system consists of a collection of an *arbitrary* but *finite* number of identical processes interacting via synchronous or asynchronous communication. Many interesting systems are of this form, for example, mutual exclusion algorithms for an arbitrary number of processes wanting to use a common resource. Many distributed programs, in particular those that control communication and synchronization of networks, also have a parallel composition of many identical processes as their body.

A challenging problem is to provide a method for the uniform verification of such programs, i.e. prove by a *single* proof that the system is correct for any value of the parameter. The key to such a uniform treatment is parameterization, i.e. presenting a single syntactic object that actually represents a family of objects.

The ability to conduct a uniform verification of a parameterized program is one of the striking advantages of the deductive method for temporal verification over algorithmic techniques such as model-checking techniques. Let n denote an arbitrary but finite number of identical processes. Model-checking can be used to verify the desired properties of the systems for specific values of n, such as $n = 3, 4, 5$. Usually, the model checker's memory capacity is exceeded for values of n smaller than 100 [80]. Furthermore, in the general case, nothing can be concluded about the property holding for any value of n from the fact that it holds for some finite set of values. In comparison, the deductive method establishes in one fell swoop the validity of the property for any value of n [80, 45].

Predicate diagrams integrate the deductive and algorithmic verification techniques. We have successfully applied our approach on the discrete and real-time systems. The using of predicate diagrams in the verification of parameterized systems will be studied in this chapter. In this work, we restrict on the parameterized systems which are *interleaving* and consist of finitely, but arbitrarily, *discrete* components. Two classes of properties will be considered, namely the properties related to the whole processes and the ones related to a single process in the systems. The latter class is sometimes called the *universal* property. For example, given a parameterized system which consists of n processes and some property P, the universal properties are expressed as formulas of the form $\forall\, k \in 1..n \,:\, P(k)$.

We start this chapter by investigating the specification of parameterized systems. Then in the next Section we present the Tickets protocol taken from [18]. In the following section, Section 7.4, we explain the verification properties of the Tickets protocol that are related to the whole processes using the ordinary predicate diagrams. In Section 7.5 we give the formal definition of parameterized predicate diagrams and verify the property of a single process in the Tickets protocol using parameterized predicate diagrams. In the end of this chapter we discuss our approach and compare to some other work.

7.2 Specification

We now investigate how to specify these systems in our modeling language. In the whole discussion, M denotes a finite and non-empty set of processes running in the system being considered.

Recall that the formula for expressing a specification is a formula of the form:

$$Spec \equiv Init \wedge \Box[Next]_v \wedge L.$$

As mentioned before, in the context of parameterized systems, the systems consist of many identical processes, precisely, they consist of the same transitions and the same liveness properties. Thus, using parameterized notation, this class of systems can be expressed as a formula of the form:

$$parSpec \equiv \forall k \in M : Init(k) \wedge \Box[Next(k)]_{v[k]} \wedge L(k). \tag{7.1}$$

In this work we only consider the parameterized discrete systems where the liveness conditions are expressed as fairness properties. Furthermore, since we also restrict to the interleaving systems, which are systems in which

each step can be attributed to exactly one process, Formula 7.1 can be expressed as follows:

$$parSpec \equiv Init \wedge \Box[\exists k \in M : Next(k)]_v \wedge \forall k \in M : L_f(k) \qquad (7.2)$$

where *Init* represents the global initial condition of the system and v is the tuple formed by all $v[k]$. From now on we will use Formula 7.2 as the standard specification form for parameterized systems.

7.3 Tickets protocol: a case study

The Tickets protocol is a mutual exclusion protocol designed for multi-client systems operating on a shared memory. In order to access the critical section, every client executes the protocol based on the first-in first-served access policy. The protocol works as follows. Initially all clients are thinking, while t and s store the same initial value. When requesting the access to the critical section, a client stores the value of the current ticket t in its local variable a. A new ticket is then emitted by incrementing t. Clients wait for their turn until the value of their local variable a is equal to the value of s. After the elaboration inside the critical section, a process releases it and the current turn is updated by incrementing s. During the execution the global state of the protocol consists of the internal state (current value of the local variable) of each process together with the current value of s and t.

Our objective is to prove that the protocol satisfies the following properties:

1. Mutual exclusion. Every time there is only maximal one process in its critical section.

2. Communal accessibility. If there exist some processes wishing to enter their critical sections then eventually there exist some processes that are in their critical sections.

3. Individual accessibility. Every time a process requests to enter its critical section, it will be eventually allowed to enter its critical section.

We can categorize those properties into two classes of properties, namely properties that are related to the whole processes and to a single process (universal properties). Mutual exclusion and communal accessibility are in the first class whereas individual accessibility is in the second class.

The specification for the Tickets protocol is given in Figure 7.1.

module TICKETS

Specification

$$
\begin{array}{rcl}
\mathit{Init} & \equiv & \forall\, k \in M \,:\, pc[k] = 1 \wedge a[k] = 0 \wedge s = 1 \wedge t = 1 \\
\mathit{Request}(k) & \equiv & \wedge\ pc[k] = 1 \wedge pc' = [pc \text{ EXCEPT } !\, k = 2] \\
 & & \wedge\ a' = [a \text{ EXCEPT } !\, k = t] \wedge t' = t + 1 \wedge s' = s \\
\mathit{Grant}(k) & \equiv & \wedge\ pc[k] = 2 \wedge a[k] = s \wedge pc' = [pc \text{ EXCEPT } !\, k = 3] \\
 & & \wedge\ s' = s \wedge t' = t \wedge a' = a \\
\mathit{Release}(k) & \equiv & \wedge\ pc[k] = 3 \wedge pc' = [pc \text{ EXCEPT } !\, k = 1] \wedge s' = s + 1 \\
 & & \wedge\ a' = a \wedge t' = t \\
v & \equiv & \langle pc, a, s, t \rangle \\
\mathit{Next}(k) & \equiv & \mathit{Request}(k) \vee \mathit{Grant}(k) \vee \mathit{Release}(k) \\
L(k) & \equiv & \wedge\ \mathrm{WF}_v(\mathit{Grant}(k)) \\
 & & \wedge\ \mathrm{WF}_v(\mathit{Release}(k)) \\
\mathit{Tickets} & \equiv & \mathit{Init} \wedge \Box[\exists\, k \in M \,:\, \mathit{Next}(k)]_v \wedge \forall\, k \in M \,:\, L(k)
\end{array}
$$

Theorem

1. $\mathit{Tickets} \rightarrow \Box(\forall\, i, j \in M \,:\, pc[i] = 3 \wedge pc[j] = 3 \rightarrow i = j)$
2. $\mathit{Tickets} \rightarrow \Box(\exists\, k \in M \,:\, pc[k] = 2 \rightarrow \Diamond(\exists\, k \in M \,:\, pc[k] = 3))$
3. $\mathit{Tickets} \rightarrow \forall\, k \in M \,:\, \Box(pc[k] = 2 \rightarrow \Diamond pc[k] = 3)$

Figure 7.1: Tickets protocol for $n \geq 1$ processes.

7.4 Verification using predicate diagrams

In this section we will study the use of predicate diagrams in the verification of parameterized systems. Recalling the definition of predicate diagrams, Definition 5.2, predicate diagrams are defined relatively to two sets, namely a set of state predicates $\mathcal{P}$ and a set of the (names of) actions $\mathcal{A}$. In the context of parameterized systems we can say that $\mathcal{A}$ now contains the (names of) the parameterized actions.

A run of a predicate diagram is now an ω-sequence of $\rho = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ such that for every $i \in \mathbb{N}$, s_i is a state, n_i is a node and $A_i \in \{\tau\} \cup \mathcal{A}$ (where A_i is a parameterized action) such that all the conditions in Definition 5.3 are satisfied.

It is already explained in Chapter 5 that verification process using predicate diagrams is done in two steps. The first step is to find a predicate diagram that can be proven to be the correct representation of the system to be verified, i.e. the diagram conforms to the system specification. For proving whether a diagram conforms to a specification or not, the conformance theorem, Theorem 5.4 is used. With the current setting, i.e. the using of parameterized actions, some modifications should be done over the Theorem 5.4. In particular, the conditions related to the fairness conditions should be

treated slightly differently from non-parameterized ones.

We need to address one important issue that will be used later, which is the issue about fairness. Note that in the specification the fairness condition is represented as a conjunction of formulas of the forms $\forall\, k \in M : \mathrm{WF}_v(A(k))$ and/or $\forall\, k \in M : \mathrm{SF}_v(A(k))$, i.e. for every process k in M and for some parameterized action $A(k)$, we associate weak and strong fairness, respectively, with $A(k)$. Let's turn to the definition of predicate diagrams, in particular the definition of ζ. In the context of parameterized systems, $\zeta : \mathcal{A} \to \{\mathrm{NF}, \mathrm{WF}, \mathrm{SF}\}$ is now a mapping that associates a fairness condition with every parameterized action $A(k)$ in $\mathcal{A}$. For example, for some parameterized action $A(k)$, if $\zeta(A(k))$ then we mean that $\mathrm{WF}_v(\exists\, k \in M : A(k))$.

We say that a predicate diagram G *conforms* to a parameterized program *parSpec* if every behavior that satisfies *parSpec* is a trace through G.

Theorem 7.1 *Let $G = (N, I, \delta, o, \zeta)$ be a predicate diagram over $\mathcal{P}$ and $\mathcal{A}$ and let $parSpec \equiv Init \wedge \Box[\exists\, k \in M : Next(k)]_v \wedge \forall\, k \in M : L_f(k)$ be a parameterized system. If all the following conditions hold then G conforms to parSpec:*

1. $\models Init \to \bigvee_{n \in I} n.$
2. $\approx n \wedge [\exists\, k \in M : Next(k)]_v \to n' \vee \bigvee_{(m, A(k)) : (n,m) \in \delta_{A(k)}} \langle \exists\, k \in M : A(k) \rangle_v \wedge m'.$
3. *For all $n, m \in N$ and all $(t, \prec) \in o(n, m)$*
   (a) $\approx n \wedge m' \wedge \bigvee_{A(k) : (n,m) \in \delta_{A(k)}} \langle \exists\, k \in M : A(k) \rangle_v \to t' \prec t.$
   (b) $\approx n \wedge [\exists k \in M : Next(k)]_v \wedge n' \to t' \preceq t.$
4. *For every action $A(k) \in \mathcal{A}$ such that $\zeta(A(k)) \neq \mathrm{NF}$*
   (a) *If* $\zeta(A(k)) = \mathrm{WF}$ *then* $\models parSpec \to \mathrm{WF}_v(\exists\, k \in M : A(k))$.
   (b) *If* $\zeta(A(k)) = \mathrm{SF}$ *then* $\models parSpec \to \mathrm{SF}_v(\exists\, k \in M : A(k))$.
   (c) $\approx n \to$ ENABLED $\langle \exists\, k \in M : A(k) \rangle_v$ *holds whenever* $n \in En(A(k))$.
   (d) $\approx n \wedge \langle \exists\, k \in M : A(k) \rangle_v \to \neg m'$ *holds for all* $n, m \in N$ *such that* $(n, m) \notin \delta_{A(k)}$.

Proof. (sketch) This theorem is a direct consequence of Theorem 5.4. Notice that $\exists k \in M \,:\, A_1(k) \vee \ldots \vee A_n(k)$ is equivalent to $(\exists k \in M \,:\, A_1(k)) \vee \ldots \vee (\exists k \in M \,:\, A_n(k))$. Thus, we can use the proof for Theorem 5.4 which is given in [26] and [83] as reference in order to prove Theorem 7.1. ■

Figure 7.2 depicts a suitable predicate diagram for the Tickets protocol where *cs* represents a set of processes whose *pc* is equal to 3, i.e. $pc_1 = \{k \in M \,:\, pc[k] = 1\}$ and $cs = \{k \in M \,:\, pc[k] = 3\}$.

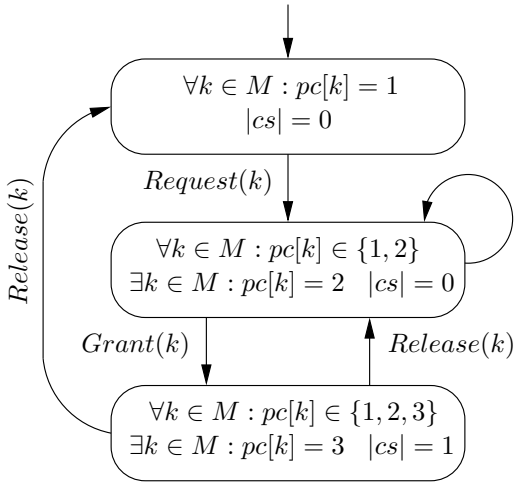

Figure 7.2: Predicate diagram for the Tickets protocol for $n \geq 1$ processes.

Although there exists a self-loop on the middle node, we don't need ordering annotations for avoiding an infinite loop. Insteads we rely on the fairness assumption for the $Grant(k)$ action to prove the communal accessibily property.

Using theorem 7.1 we can prove that this diagram conforms to the *Tickets* specification. For example, we have:

- $Init \rightarrow \forall\, k \in M \,:\, pc[k] = 1 \wedge |cs| = 0.$
- $\forall k \in M \,:\, pc[k] = 1 \wedge |cs| = 0 \wedge [\exists\, k \in M \,:\, Next(k)]_v \longrightarrow$
  $\vee\ \forall k \in M \,:\, pc[k]' = 1 \wedge |cs'| = 0$
  $\vee\ \wedge\ \langle \exists\, k \in M : Request(k) \rangle_v$
  $\wedge\ \forall k \in M \,:\, pc[k]' \in \{1,2\} \wedge \exists k \in M \,:\, pc[k]' = 2 \wedge |cs'| = 0.$
- $\wedge\ \forall k \in M \,:\, pc[k] \in \{1,2\} \wedge \exists\, k \in M \,:\, pc[k] = 2 \wedge |cs| = 0$
  $\wedge\ \forall k \in M \,:\, pc[k]' \in \{1,2\} \wedge \exists\, k \in M \,:\, pc[k]' = 2 \wedge |cs'| = 0$
  $\wedge\ \langle \exists\, k \in M : Request(k) \rangle_v$

$\longrightarrow |pc_1'| < |pc_1|$.

Predicate diagram in Figure 7.2 can be used to prove the mutual exclusion and communal accessibility properties of the Tickets protocol. We can model-check the diagram, for example with SPIN, as explained in [26, 83].

7.5 Parameterized predicate diagrams

So far, we have successfully applied predicate diagrams in proving the mutual exclusion and communal accessibility properties of the Tickets protocol. Unfortunately, this approach cannot be used to prove the individual accessibility of that protocol. In general, this approach suffers from the limitation that it cannot be used for the verification of universal properties.

In proving universal properties we have to find a way that enables us to keep track the behaviors of some particular process. The idea is to view the systems as collections of two components, which are a particular process and the collection of the rest of the processes.

Given a parameterized system specification *parSpec* and a property P, our goal is to prove the validity of $parSpec \rightarrow \forall\, k \in M \,:\, P(k)$. Let $i \in M$ be some process. We reduce the proof to the proof of $parSpec \wedge i \in M \rightarrow P(i)$[1]. If the proof succeeds then, since we apply the standard quantifier introduction rule of first-order logic, we can conclude that the property holds over each process in the system, i.e. $\forall\, k \in M \,:\, P(k)$ is valid.

For the sake of the presentation, in the following we will denote by $A(i)$ for some action of process i and $A(\mathbf{k})$ for formula $\exists\, k \in M \setminus \{i\} \,:\, A(k)$ for some action of any process other than i.

Definition 7.2 *(quantified-actions) For a set of parameterized actions $\mathcal{A}$, we denote by $\Phi(\mathcal{A})$, the set of quantified-actions which are formulas of the form $A(i)$ or $A(\mathbf{k})$ for $A(k)$ some parameterized action in $\mathcal{A}$.*

We now define a variant of predicate diagrams that can be used for verifying universal properties of parameterized systems. We call this variant *parameterized predicate diagrams*, or PPDs for short.

Definition 7.3 *(PPD) Given a set of state predicates $\mathcal{P}$, a set of parameterized actions $\mathcal{A}$ and the set of quantified-actions over parameterized actions in $\mathcal{A}$, $\Phi(\mathcal{A})$, PPD over $\mathcal{P}, \mathcal{A}$, and $\Phi(\mathcal{A})$, $\mathcal{G}$, is given by a tuple (N, I, δ, o, ζ) where*

[1] We call such method *Skolemization*.

- *N, I and o as defined in Definition 5.1.*
- *a family $\delta = (\delta_B)_{B \in \Phi(\mathcal{A})}$ of relations $\delta_B \subseteq N \times N$; we also denote by δ the union of the relations δ_B, for $B \in \Phi(\mathcal{A})$ and write $\delta_=$ to denote the reflexive closure of the union of these relations and*
- *a mapping $\zeta : \mathcal{A} \to \{\mathrm{NF}, \mathrm{WF}, \mathrm{SF}\}$ that associates a fairness condition with every parameterized action in $\mathcal{A}$; the possible values represent no fairness, weak fairness, and strong fairness.*

We say that the quantified-action $B \in \Phi(\mathcal{A})$ can be taken at node $n \in N$ iff $(n, m) \in \delta_B$ holds for some $m \in N$, and denote by $En(B) \subseteq N$ the set of nodes where B can be taken.

Instead of using parameterized actions, we now use quantified-actions as edge labels. On the contrary, we still associate the fairness annotations with parameterized actions and not with quantified-actions for ensuring that for some parameterized action $A(k)$, the quantified actions of $A(k)$ have the same fairness conditions.

Definition 7.4 *Let $\mathcal{G} = (N, I, \delta, o, \zeta)$ be a* PPD *over $\mathcal{P}, \mathcal{A}$ and $\Phi(\mathcal{A})$. A run of $\mathcal{G}$ is an ω-sequence $\rho = (s_0, n_0, A_0)\ (s_1, n_1, A_1) \ldots$ of triples where s_i is a state, $n_i \in N$ is a node and $A_i \in \Phi(\mathcal{A}) \cup \{\tau\}$ is an action such that all of the following conditions hold:*

1. *$n_0 \in I$ is an initial node.*
2. *$s_i[\![n_i]\!]$ holds for all $i \in \mathbb{N}$.*
3. *For all $i \in \mathbb{N}$ either $A_i = \tau$ and $n_i = n_{i+1}$ or $A_i \in \Phi(\mathcal{A})$ and $(n_i, n_{i+1}) \in \delta_{A_i}$.*
4. *If $A_i \in \Phi(\mathcal{A})$ and $(t, \prec) \in o(n_i, n_{i+1})$, then $s_{i+1}[\![t]\!] \prec s_i[\![t]\!]$.*
5. *If $A_i = \tau$ then $s_{i+1}[\![t]\!] \preceq s_i[\![t]\!]$ holds whenever $(t, \prec) \in o(n_i, m)$ for some $m \in N$.*
6. *For every quantified-action $B \in \Phi(\mathcal{A})$ of parameterized action $A(k) \in \mathcal{A}$ such that $\zeta(A(k)) = \mathrm{WF}$ there are infinitely many $i \in \mathbb{N}$ such that either $A_i = B$ or $n_i \notin En(B)$.*
7. *For every action $B \in \Phi(\mathcal{A})$ of parameterized action $A(k) \in \mathcal{A}$ such that $\zeta(A(k)) = \mathrm{SF}$, either $A_i = B$ holds for infinitely many $i \in \mathbb{N}$ or $n_i \in En(B)$ holds for only finitely many $i \in \mathbb{N}$.*

We write runs($\mathcal{G}$) to denote the set of runs of $\mathcal{G}$.

The set tr($\mathcal{G}$) traces through $\mathcal{G}$ consists of all behaviors $\sigma = s_0 s_1 \ldots$ such that there exists a run $\rho = (s_0, n_0, A_0)(s_1, n_1, A_1)\ldots$ of $\mathcal{G}$ based on the states in σ.

We say that a PPD $\mathcal{G}$ *conforms* to a parameterized system *parSpec* if every behavior that satisfies *parSpec* is a trace through $\mathcal{G}$.

Theorem 7.5 *Let $\mathcal{G} = (N, I, \delta, o, \zeta)$ be a* PPD *$\mathcal{P}, \mathcal{A}$ and $\Phi(\mathcal{A})$ as defined and let parSpec $\equiv$ Init $\wedge \Box[\exists\, k \in M : Next(k)]_v \wedge \forall\, k \in M : L_f(k)$ be a parameterized system. If all the following conditions hold then $\mathcal{G}$ conforms to parSpec:*

1. $\models Init \rightarrow \bigvee_{n \in I} n.$
2. $\models\!\!\approx n \wedge [\exists\, k \in M : Next(k)]_v \rightarrow n' \vee \bigvee_{(m,B):(n,m)\in\delta_B} \langle B \rangle_v \wedge m'$
3. *For all $n, m \in N$ and all $(t, \prec) \in o(n, m)$*
   (a) $\models\!\!\approx n \wedge m' \wedge \bigvee_{B:(n,m)\in\delta_B} \langle B \rangle_v \rightarrow t' \prec t$
   (b) $\models\!\!\approx n \wedge [\exists k \in M : Next(k)]_v \wedge n' \rightarrow t' \preceq t.$
4. *For every parameterized action $A(k) \in \mathcal{A}$ such that $\zeta(A(k)) \neq$ NF*
   (a) *If $\zeta(A(k)) =$ WF then for every quantified action B of $A(k)$, $\models$ parSpec $\rightarrow \mathrm{WF}_v(B)$.*
   (b) *If $\zeta(A(k)) =$ SF then for every quantified action B of $A(k)$, $\models$ parSpec $\rightarrow \mathrm{SF}_v(B)$.*
   (c) $\models\!\!\approx n \rightarrow \langle$ ENABLED $B\rangle_v$ *holds for every quantified action of $A(k)$, B, whenever $n \in En(B)$.*
   (d) $\models\!\!\approx n \wedge \langle B \rangle_v \rightarrow \neg m'$ *holds for all $n, m \in N$ and for every quantified action of $A(k)$, B, such that $(n, m) \notin \delta_B$.*

Proof. (sketch) Like theorem 7.1, this theorem is again a consequence of Theorem 5.4. The proofs of conditions $1-3$ are similar to the proof of conditions $1-3$ Theorem 7.1. For the proof of condition 4, we have to consider two cases, namely for the quantified actions of the form $A(i)$ and for the one of the form $A(\mathbf{k})$. ■

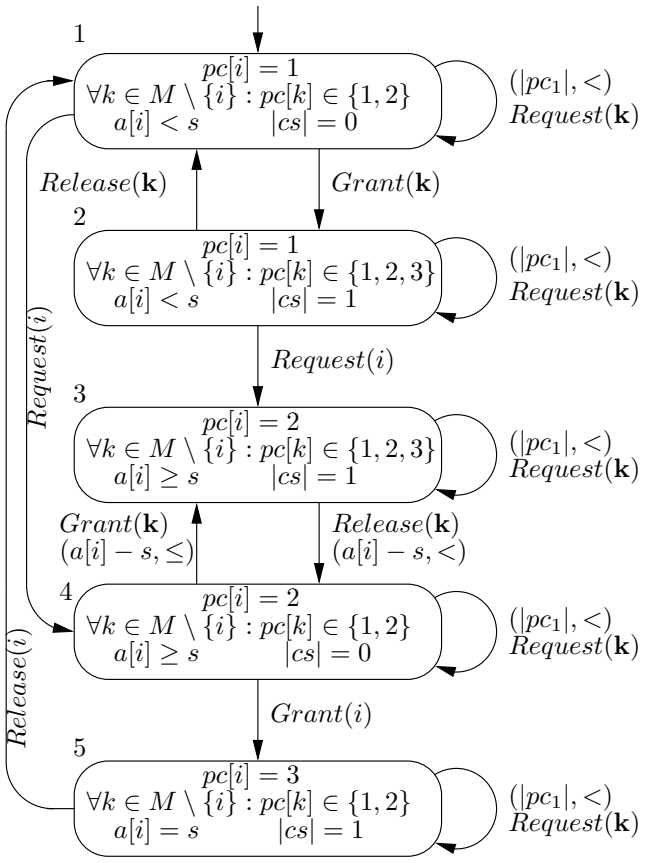

Figure 7.3: PPD for Tickets protocol for $n \geq 1$ processes.

Let pc_1 and cs be the set of processes whose pc is equal to 1 and 3, respectively, the diagram in Figure 7.3 is a suitable PPD for Tickets protocol for $n \geq 1$ processes. We associate an ordering annotation $(|pc_1|, <)$ with the loop of every node to ensure that eventually the system will leave the loops due to the finiteness of M. We also associate the ordering annotation $(a[i] - s, <)$ with the edge from node 4 to node 3 and associate the ordering annotation $(a[i] - s, \leq)$ with the edge from node 3 to node 4 for avoiding loops that may happen between the pair of nodes. Thus, we can ensure that eventually process i is allowed to enter its critical section. The choice of the orderings is based on the fact that whenever $pc[i] = 2$ then the difference between $a[i]$ and s is decreased whenever some process other than i leaves its critical section and increments the value of s by 1.

The PPD in Figure 7.3 conforms to the *Tickets* specification in Figure 7.1. We can use Theorem 7.5 for proving this conformance. For example we have:

- $Init \rightarrow \left(\begin{array}{l} pc[i] = 1 \\ \wedge\ \forall k \in M \setminus \{i\} \ :\ pc[k] \in \{1,2\} \\ \wedge\ a[i] < s \\ \wedge\ |cs| = 0 \end{array} \right).$

- $$\left(\begin{array}{l} pc[i] = 1 \\ \wedge\ \forall k \in M \setminus \{i\} \ :\ pc[k] \in \{1,2\} \\ \wedge\ a[i] < s \\ \wedge\ |cs| = 0 \end{array} \right) \wedge [\exists\, k \in M \ :\ Next(k)]_v \rightarrow$$
  $$\vee \left(\begin{array}{l} pc[i]' = 1 \\ \wedge\ \forall k \in M \setminus \{i\} \ :\ pc[k]' \in \{1,2\} \\ \wedge\ a[i]' < s \\ \wedge\ |cs'| = 0 \end{array} \right)$$
  $$\vee \langle Request(i) \rangle_v \wedge \left(\begin{array}{l} pc[i]' = 2 \\ \wedge\ \forall k \in M \setminus \{i\} \ :\ pc[k]' \in \{1,2\} \\ \wedge\ a[i]' \geq s \\ \wedge\ |cs'| = 0 \end{array} \right)$$
  $$\vee \langle \exists\, k \in M \setminus \{i\} : Request(k) \rangle_v \wedge \left(\begin{array}{l} pc[i]' = 1 \\ \wedge\ \forall k \in M \setminus \{i\} : pc[k]' \in \{1,2\} \\ \wedge\ a[i]' < s \\ \wedge\ |cs'| = 0 \end{array} \right)$$
  $$\vee \langle \exists\, k \in M \setminus \{i\} : Grant(k) \rangle_v \wedge \left(\begin{array}{l} pc[i]' = 1 \\ \wedge\ \forall k \in M \setminus \{i\} : pc[k]' \in \{1,2,3\} \\ \wedge\ a[i]' < s \\ \wedge |cs'| = 1 \end{array} \right).$$

- $$\left(\begin{array}{l} \wedge\ \wedge\ pc[i] = 2 \\ \quad \wedge\ \forall k \in M \setminus \{i\} \ :\ pc[k] \in \{1,2,3\} \\ \quad \wedge\ a[i] \geq s \\ \quad \wedge\ |cs| = 1 \\ \wedge\ \wedge\ pc[i]' = 2 \\ \quad \wedge\ \forall k \in M \setminus \{i\} \ :\ pc[k]' \in \{1,2\} \\ \quad \wedge\ a[i]' \geq s \\ \quad \wedge\ |cs'| = 0 \\ \wedge\ \langle \exists\, k \in M \setminus \{i\} \ :\ Release(k) \rangle_v \end{array} \right) \longrightarrow (a[i]' - s') < (a[i] - s).$$

Using the diagram in Figure 7.3 we can prove that it is always the case that whenever process i request to enter its critical section, it will eventually enters its critical section, i.e. we can prove the validity of formula

$Tickets \rightarrow \Box(pc[i] = 2 \rightarrow \Diamond pc[i] = 3)$. Moreover, since we have just applied the standard quantifier introduction rule of first-order logic, this implies the validity of formula $\forall\, k \in M : Tickets \rightarrow \Box(pc_k = 2 \rightarrow \Diamond pc_k = 3)$ as required.

Note that the mutual exclusion and communal accessibility properties of Tickets protocol can also be verified using the PPD in Figure 7.3. Thus, in general, PPD can not only be used to prove the universal properties of a parameterized system, but also to verify the properties that related to the whole system.

7.6 Discussion and related work

Verification of parameterized systems is often done by hand, or with the guidance of a theorem prover [85, 80, 54]. Several methods have been proposed that, to various degrees, automate this verification process. Methods based on manual construction of a process invariant are proposed in [30, 107]. However, as the general problem is undecidable [12], it is not in general possible to obtain a finite-state process invariant. For classes of parameterized systems obeying certain constraints, for example [50, 44], there exists algorithms for model checking the parameterized systems.

In this work we have restricted to a class of parameterized systems that are interleaving and consist of a finitely, but arbitrarily, discrete components. The parameterized systems are represented as parameterized TLA specifications. The verification is done deductively and algorithmically by means of diagrams. Our diagrams can be viewed as the abstract representation of parameterized systems, i.e. we represent a family of processes in a single diagram. The same spirit but using difference formalism is the work from Baukus et.al.[13, 14]. They propose a method for the verification of universal properties of parameterized networks based on the transformation of an infinite family of systems into a single WS1S [23, 103] transition system and applying abstraction techniques on this system.

By using the Tickets protocol as a running example, we have shown that with a little modification on the conformance theorem of (ordinary) predicate diagram, it is possible to verify properties related to whole processes with predicate diagrams. In order to verify the universal properties we define a variant of predicate diagrams called parameterized predicate diagrams or PPDs.

For handling the universal properties we distinguish some single arbitrary process from the rest of processes. This can be extended for proving the prop-

erties that related to some set of particular processes. The idea is to consider those processes separately from the rest of the processes. In this case, some more complex reasoning might be necessary to do, such as induction on the number of processes, depending on the protocol at hand.

We have shown that PPDs can be used to prove the universal properties and the properties related to the whole processes as well. In contrast to the ordinary predicate diagrams, if we work with PDDs then we have to consider the actions of some particular process separately from the actions of the rest of the processes. Therefore, in the worst case, the number of the generated proof obligations is twice as the number of proof obligations generated by the ordinary predicate diagrams.

Chapter 8

Generation of diagrams

8.1 Overview

So far we have described the use of predicate diagrams, TPDs and PPDs, in the verification of reactive systems. If we recall the verification process using those diagrams, there are two steps should be done. The first step is to find a diagram that conforms to the specification and then the second step is to prove that the diagram satisfies the property to be verified. In order to prove that a diagram really represents the specification we equip every type of diagram with a theorem, which we call conformance theorem. For example, the conformance theorem of predicate diagram, Theorem 5.4, can be used to justify that a given predicate diagram conforms to a given specification. The problem is that it may generate a number of proof obligations that is quadratic in the number of nodes. It is therefore desirable to construct predicate diagrams semi-automatically from a specification whenever this is possible.

In our methodology diagrams can be viewed as the abstract representations of the systems being considered. Thus, the problem can be rephrased as the generation of abstract representation of systems. There is much work on the generation of abstract systems [31, 37, 74, 36], usually based on the ideas of *abstract interpretation* [34]. We will follow the approach proposed by CANSELL et al.[26].

The contribution of this chapter is the development of two prototype tools, namely PreDiaG and parPreDiaG, that can be used for generating predicate diagrams and PPDs.

We first describe the concept of our first implementation, namely the generation of predicate diagrams. We then move to the generation of the PPD in the next section. A short discussion and related work will be given

at the end of this chapter.

8.2 Generation of predicate diagrams

Given a discrete system's specification $Spec \equiv Init \wedge \Box[Next]_v \wedge L_f$, a set of predicates, $\mathcal{P}$ and a set of actions $\mathcal{A}$, the goal of our tool is to generate a predicate diagram over $\mathcal{P}$ and $\mathcal{A}$ that conforms to the safety part of the specification. In other word, our objective is to find a predicate diagram $G = (N, I, \delta, \emptyset, \emptyset)$ such that G conforms to $Init \wedge \Box[Next]_v$. Notice that we don't consider liveness properties in our implementation.

8.2.1 Nodes

By definition, nodes of predicate diagrams are sets of $\overline{\mathcal{P}}$ which are interpreted conjunctively. Therefore, we may assume a node to be a conjunction of literals. In the implementation, nodes are represented as n-bit vectors where n is the number of state predicates in $\mathcal{P}$. For example, if $\mathcal{P}$ consists of n state predicates $p_1, \ldots, p_n$, then a node will be represented as $b_1 \ldots b_n$ such that for every $i \in 1..n$, either $b_i = 1$ if p_i holds or $b_i = 0$ if p_i doesn't hold on that node.

For a set of state predicates $\mathcal{P}$, we denote by $Nodes(\mathcal{P})$ the set containing all nodes or n-bit vectors formed by the state predicates of $\mathcal{P}$.

8.2.2 Abstract interpretation

We have used the concept of *abstract interpretation* [34] in the construction of predicate diagrams. The relationship between concrete and abstract states that underlies abstract interpretation is traditionally described by a Galois connection, which is defined as follows.

Definition 8.1 *(Galois connection) Let $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ be partially-ordered sets (posets) and let $\alpha : L_1 \rightarrow L_2$ and $\gamma : L_2 \rightarrow L_1$ be functions. The pair (α,γ) is said to form a Galois connection if*

$$\forall\, x \in L_1, y \in L_2 : x \sqsubseteq_1 \gamma(y) \leftrightarrow y \sqsubseteq_2 \alpha(x).$$

The set L_1 and L_2 are called the concrete and abstract domain, respectively. The function α is called the abstraction function and γ is called the concretization function.

Graphically, we can denote a Galois connection, for example, as is shown in Figure 8.1.

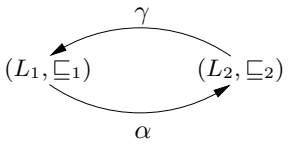

Figure 8.1: Galois connection between $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$.

In our setting, we choose the powerset of "concrete" states, 2^{Σ}, as L_1 and $\mathit{Nodes}(\mathcal{P})$ as L_2. The abstraction function returns a set of nodes such that

$$\alpha(S) = \{n \in \mathit{Nodes}(\mathcal{P}) : \forall s \in S : s \in \gamma(n)\},$$

where $S \subseteq \Sigma$ and the concretization function γ produces a set of states that are models of a node

$$\gamma(n) = \{s \in \Sigma : s \models n\}.$$

8.2.3 Abstract evaluation of an action

Given an abstract state $n \in \mathit{Nodes}(\mathcal{P})$, the main problem is to compute an abstract representation of the set of successor states of the states in $\gamma(n)$ with respect to some action A of the given specification.

Definition 8.2 *For $m, n \in \mathit{Nodes}(\mathcal{P})$ and an action $A \in \mathcal{A}$, we say that n is an abstract successor of m iff for all states s, t in Σ, if $s \in \gamma(m)$ and $s[\![A]\!]t$ then $t \in \gamma(n)$.*

Because we may alternatively interpret abstract states as elements of L_2 and as predicates over concrete states, the above definition can be restated as requiring

$$\models m \wedge A \rightarrow n'.$$

Since m is a node, we may assume m to be a conjunction of literals, and try to compute some successors n in disjunctive normal form.

The abstract evaluation of A from m is done in two steps. The first step is to check whether A can be taken on m or not. This step is done by evaluating those sub-formulas of A that contain only unprimed variables to either **true** or **false** in order to simplify the action formula. The second step is to evaluate as many formulas P', for $P \in \mathcal{P}$, as possible in order to assemble information about the predicates that are **true** or **false** after the action has been executed.

As example, let's take the small system called AnyY problem [58]. Figure 8.2 presents a simple program consisting of two processes communicating

by the shared variable x, which is initially set to 0. Process P_1 keeps incrementing variable y as long as $x \neq 0$. Once process P_2 sets x to 1, process P_2 terminates and some time later so does P_1 as soon as it observes that $x = 1$. We will prove the termination of this system, i.e. eventually x is equal to 1, and that the values of y is never negative.

module ANYY

Specification

$$
\begin{array}{rcl}
\mathit{Init} & \equiv & x = 0 \wedge y = 0 \\
P_1 & \equiv & x = 0 \wedge y' = y + 1 \wedge x' = x \\
P_2 & \equiv & x = 0 \wedge x' = 1 \wedge y' = y \\
v & \equiv & \langle x, y \rangle \\
\mathit{Next} & \equiv & P_1 \vee P_2 \\
\mathit{AnyY} & \equiv & \mathit{Init} \wedge \Box[\mathit{Next}]_v
\end{array}
$$

Theorem

1. $\mathit{AnyY} \rightarrow \Diamond(x = 1)$.
2. $\mathit{AnyY} \rightarrow \Box(y \geq 0)$.

Figure 8.2: Module ANYY.

We first describe the abstract evaluation procedure for monadic predicates. If $\mathcal{P}$ containes k monadic predicates $P_1(x), \ldots, P_k(x)$ that contain the same (concrete) state variable x, we let the abstract states contain k state variables $\overline{x_1}, \ldots, \overline{x_k}$ such that $\overline{x_i}$ takes values P_i or *not-*P_i. For example, in the context of AnyY problem, the predicate $x = 0$ might be represented by the variable $\overline{x} \in \{0, 1\}$ and the predicate $y = 0$ might be represented by the variable $\overline{y} \in \{\mathit{zero}, \mathit{pos}\}$. Now every unprimed occurrence of variable x and y in A is replaced by the value assigned to $\overline{x_i}$ and $\overline{y_i}$ by the abstract source state m, for every $\overline{x_i}$ and $\overline{y_i}$ that appears in m. The formula simplification is done with help of a *rule-base*, which is a set of rewriting rules. A rewriting rule is an implication formula, such that the premise is the formula to be simplified and the conclusion is the simplified formula.

Suppose $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$, where $p_1 \equiv x = 0, p_2 \equiv x = 1, p_3 \equiv y = 0$ and $p_4 \equiv y > 0$ and $\mathcal{A} = \{P_1, P_2\}$. We will consider the evaluation of action P_1 from the state $m = \{\overline{x} = 0, \overline{y} = \mathit{zero}\}$. Assume that the set of rewriting rules include

$$
\begin{array}{rcl}
0 = 0 & \rightarrow & \mathit{true} \\
\mathit{zero} + 1 & \rightarrow & \mathit{pos} \\
\mathit{pos} + 1 & \rightarrow & \mathit{pos}
\end{array}
$$

For checking whether A can be taken on m or not, we consider the unprimed part of A, which is a sub-formula containing no primed expressions.

We say that A can be taken on m whenever the simplification process results **true**.

Now we simplify the unprimed part of P_1, which is $x = 0$, as follows:

formula	simplified by	
$\overline{x} = 0$	$0 = 0$	$(\overline{x} = 0)$
	true	$(0 = 0 \rightarrow \textbf{true})$

Since the simplification returns **true**, this means that P_1 can be taken on this state, we continue to generate the successors of this state. We now simplify the primed part of P_1 which is $\overline{y}' = \overline{y} + 1 \wedge \overline{x}' = \overline{x}$ as follows:

formula	simplified by	
$\overline{y}' = \overline{y} + 1$	$\overline{y}' = zero + 1$	$(\overline{y} = zero)$
	$\overline{y}' = pos$	$(zero + 1 \rightarrow pos)$

formula	simplified by	
$\overline{x}' = \overline{x}$	$\overline{x}' = 0$	$(\overline{x} = 0)$

Thus, the simplification process results $\overline{y}' = pos \wedge \overline{x} = 0$. The successor state of m is $\{\overline{x} = 0, \overline{y} = pos\}$.

In the implementation we use MONA [59] for generating the abstract states. Since nodes are represented as n-bit vectors, in this case every node is represented as $b_1b_2b_3b_4$ where every b_i is p_i atau $\neg p_i$. The simplication results $\overline{x} = 0$ and $\overline{y} = pos$, which imply that p_1 dan p_4 hold. To generate all n-bit vectors that satisfy this condition we give the formula $p_1 \wedge p_4$ to MONA as input. With this input, MONA will result such an expression 1XX1 which means that the values of p_2 and p_3 could either be 0 or 1. It follows that 1101 is also a valid node, which is wrong, since p_2 won't be **true** whenever p_1 is **true**, and vice versa. To avoid the tool for resulting such nodes, we add some formula called *constraint*. In the context of AnyY problem, we take $p_1 \leftrightarrow \neg p_2 \wedge p_3 \leftrightarrow \neg p_4$ as constraint. Giving the simplification's result and the constraint, we get the node 1001.

The generation of initial abstract states is done by considering *Init*. We assign value to every state predicate in $\mathcal{P}$ based on the information in *Init*. Again, in the context of AnyY problem, the predicates p_1 and p_3 are **true**. We give the formula $p_1 \wedge p_3$ plus the constraint above to MONA as input. The resulted node is 1010.

As conclusion, the generation of predicate diagrams is done by first evaluating *Init* to generate the initial abstract states. Then, for every abstract

state m and every action A in $\mathcal{A}$, the tool evaluates A from m to produce the next states of m. The tool repeats this process until there are no more new abstract states can be generated.

The resulted diagram for AnyY problem is given in Figure 8.3.

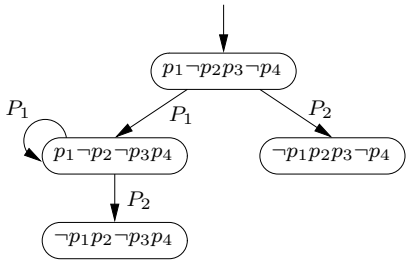

Figure 8.3: The resulted predicate diagram for AnyY problem.

8.2.4 Maybe edges

Suppose we want to generate the predicate diagram for the Bakery algorithm. The specification is given in Figure 5.3. Assume that we choose $\{0, 1, 2, 3\}$ as the abstract domain for the variables $\overline{pc_1}$ and $\overline{pc_2}$ and$\{zero, pos\}$ for the variables $\overline{t_1}$ and $\overline{t_2}$. The underlying set of rewrite rules, for example, include:

$0 = 0 \rightarrow \mathbf{true}$	$zero = zero \rightarrow \mathbf{true}$	$zero \leq zero \rightarrow \mathbf{true}$
$0 = 1 \rightarrow \mathbf{false}$	$zero = pos \rightarrow \mathbf{false}$	$zero \leq pos \rightarrow \mathbf{true}$
$\vdots$	$zero + 1 \rightarrow pos$	$pos \leq zero \rightarrow \mathbf{false}$
$4 = 4 \rightarrow \mathbf{true}$	$pos + 1 \rightarrow pos$	

Note that expressions such as $pos = pos$ or $pos \leq pos$ cannot be simplified. As consequence the evalution on action on this state is fail, since it results neither **true** nor **false**.

In such situation, following Cansell et al. [26], we use the concept of *maybe edges*, which can described as follows.

We assign the value **maybe** to such uninterpreted expression:

$$pos \leq pos \rightarrow \mathbf{maybe}.$$

Maybe values are propagated using rewrite rules such as:

$$\begin{aligned} \mathbf{maybe} \wedge \mathbf{maybe} &\rightarrow \mathbf{maybe} \\ \mathbf{false} \wedge \mathbf{maybe} &\rightarrow \mathbf{false} \\ \neg \mathbf{maybe} &\rightarrow \mathbf{maybe} \end{aligned}$$

Successor states can be extracted as described above even in the presence of maybe conjuncts, but we remember such situation and indicate edges obtained in this way using different color (red).

For the Bakery specification, we obtain the graph shown in Figure 8.4 with the maybe edges are indicated with dasched edges. (The original generated predicate diagram is given in Figure B.10). For the sake of compactness, we label every node with the predicates that hold on that node. For example, the node 10000100001010 will be labeled it with $p_1p_6p_{11}p_{14}$.

Unfortunately, the resulted diagram can not be used to prove the mutual exclusion property. As we can see, on the node $p_6p_9p_{12}p_{14}$, predicates p_4 and p_9 hold, which means that the both processes are in their critical sections. However, every path leading to the abstract state contains a dashed edge, and these edges indicate opportunities for refining the approximation by reconsidering the transition in view of the concrete specification. Some refinement techniques can be found, for example in [26] and [83].

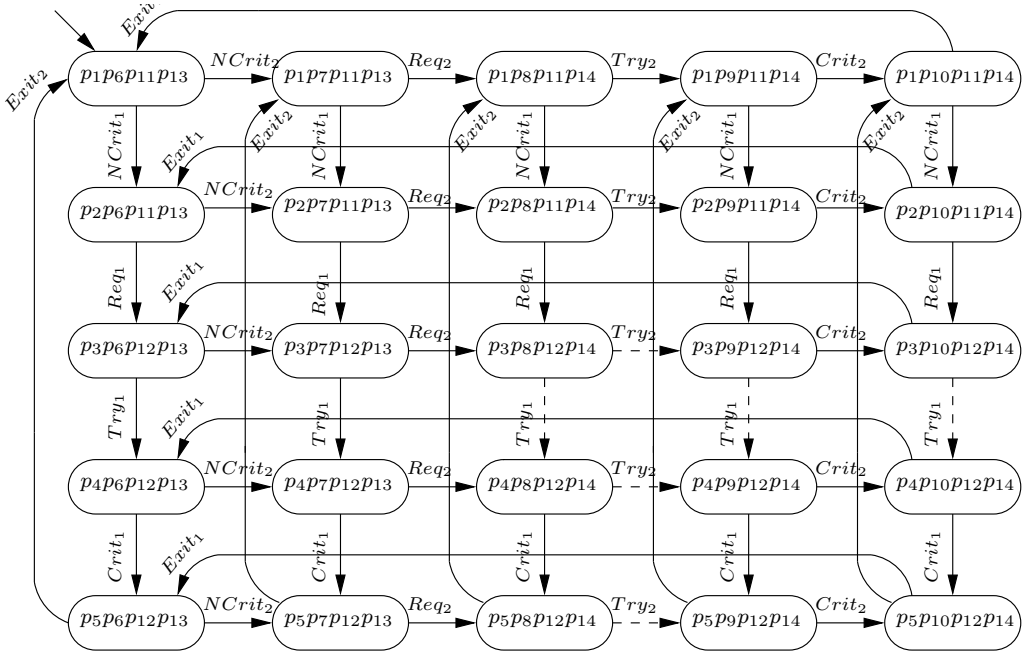

Figure 8.4: Predicate diagram for Bakery algorithm.

8.3 Generation of PPDs

We have presented the generation of predicate diagrams. We now move to the next topic, namely the generation of PPDs. We can not use Prediag for

generating PPDs, since PreDiaG cannot handle the expression with quantifications.

Given a parameterized system specification $parSpec \equiv Init \wedge \Box[\exists\, k \in M : Next(k)]_v \wedge \forall k \in M : L_f(k)$, a set of state predicates $\mathcal{P}$ and a set of parameterized actions $\mathcal{A}$, our goal is to generate a PPD that conforms to the safety part of *parSpec*.

Nodes of PPDs are represented as n-bit vectors as before. The generation process consists of finding the initial abstract states and evaluating every action in $\mathcal{A}$ from some state.

Differs from the first implementation, we now rely all the evaluation processes on MONA. We even require the specification and the predicate input files are written in the input of MONA. The translation from TLA* to the input language of MONA, unfortunately, should be done by the user. However, since the syntax of the input language of MONA is quite simple, we hope that this would not be a big problem. To learn more about the syntax of the language of MONA the readers may refer to [59]. Note that since we use MONA as a consequence we only work with formulas that can be expressed by the input language of MONA.

As illustration, let's consider the specification given in Figure 8.5. It can be proven that the specification of an abstract version of the specification of Tickets protocol for $n \geq 1$ processes in Figure 7.1. We prefer to work with the abstract version to to tackle the infiniteness problem due to the infinite domains of the variables s, t and a.

module ABSTICKETS

Specification

$$
\begin{array}{rcl}
\overline{Init} & \equiv & \forall\, k \in M : k \in Pc_1 \wedge Pc_2 = \emptyset \wedge Pc_3 = \emptyset \\
\overline{Request}(k) & \equiv & k \in Pc_1 \wedge Pc_1' = Pc_1 \setminus \{k\} \wedge Pc_2' = Pc_2 \cup \{k\} \wedge Pc_3' = Pc_3 \\
\overline{Grant}(k) & \equiv & \wedge\ k \in Pc_2 \wedge Pc_3 = \emptyset \wedge Pc_1' = Pc_1 \\
 & & \wedge\ Pc_3' = Pc_3 \wedge \{k\} \wedge Pc_2' = Pc_2 \setminus \{k\} \\
\overline{Release}(k) & \equiv & k \in Pc_3 \wedge Pc_3' = Pc_3 \setminus \{k\} \wedge Pc_1' = Pc_1 \cup \{k\} \wedge Pc_2' = Pc_2 \\
\overline{vars}(k) & \equiv & \langle Pc_1, Pc_2, Pc_3 \rangle \\
\overline{Next}(k) & \equiv & \overline{Request}(k) \vee \overline{Grant}(k) \vee \overline{Release}(k) \\
absTickets & \equiv & \overline{Init} \wedge \Box[\exists\, k \in M : \overline{Next}(k)]_{\overline{vars}}
\end{array}
$$

Figure 8.5: The Tickets protocol (abstract version).

Suppose we want to generate PPD for *absTicket* where $\mathcal{P}$ contains the following state predicates:

- $p_1 \equiv i \in Pc_1$
- $p_2 \equiv i \in Pc_2$
- $p_3 \equiv i \in Pc_3$
- $p_4 \equiv \forall\, j, k \in M \;:\; j \in Pc_3 \wedge k \in Pc_3 \rightarrow j = k$
- $p_5 \equiv Pc_3 \neq \emptyset$

and $\mathcal{A}$ contains the actions $\overline{Request}(k), \overline{Grant}(k)$ and $\overline{Release}(k)$.

The evaluation of *Init* produces one abstract initial state 10010. The process continues with the evaluation of quantified actions of every parameterized action in $\mathcal{A}$ in order to get the successor of node 10010. Consider the evaluation of quantified action $\overline{Request}(i)$ on node 10010. Giving $p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge p_4 \wedge \neg p_5$, $\overline{Request}(i)$ as input, we get a new node 01010. The evaluation of $\overline{Grant}(\mathbf{k})$ on node 10010 result a node 10011. The final diagram is given in Figure 8.6.

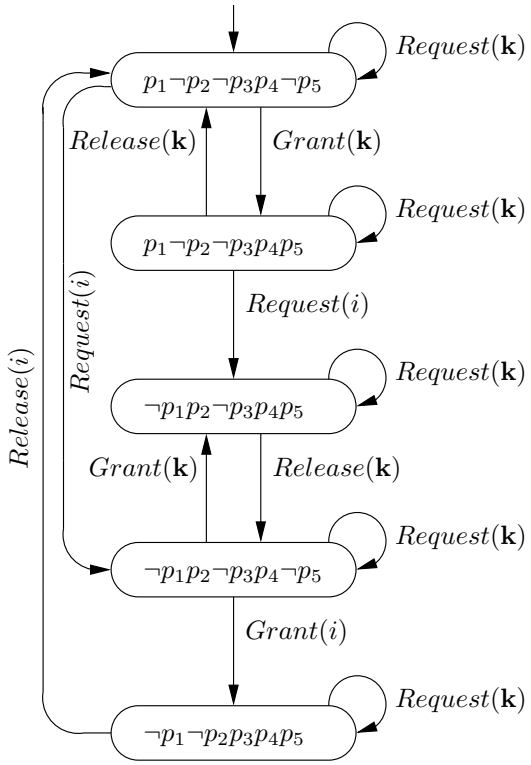

Figure 8.6: PPD for Tickets protocol with $n \geq 1$ processes.

8.4 Discussion and related work

We have presented a method for generating predicate diagrams and PPDs "semi-automatically". We use the term semi-automatically, since the user's intervention is still needed, in particular for defining the abstraction functions and rewriting rules. We have implemented this method in two prototype tools called PreDiaG and parPreDiaG.

There are some tools based on the similar idea, such as InVest from SAIDI et al. [53, 98] and SAL from BENSALEM et al. [16, 17]. PreDiaG is very closed to the tool from CANSELL et al. [26]. Instead of using the rewriting engine *Logic Solver* from Atelier B [102] and the automatic prover Simplify [42] like their implementation, PreDiaG uses its own simple rewriting engine and MONA[1]. Besides the generation algorithm that is implemented in PreDiaG, their tool has also implemented some methods for improving the abstract interpretation; but it does not support the generation of Promela code and the graphical representation of predicate diagrams. In the context of parameterized systems, parPreDiaG is inspired by PAX [91] from BAUKUS et al.

It is said that abstractions that remove too much information from the concrete system and are thus too coarse will fail to prove the property of interest. They can be refined, by adding more detail, until the property can be proved or a concrete counterexample is found. This holds also in the construction of diagrams using our tools. When the resulted diagram is too "abstract", we can refine it with details as necessary to make it more "concrete" by returning to the concrete specification.

[1]We use MONA because it is easy to use and to integrate to our tools.

Chapter 9

Conclusion and future work

We have studied the specification and verification of some classes of reactive systems, namely discrete systems, real-time systems and parameterized systems. We use TLA* from MERZ to formalize our approach.

The general formula for representing reactive systems is as follows:

$$\exists\, x : \mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L$$

where

- x is a list of internal variable,
- *Init* is a state predicate that describes the initial states,
- *Next* is an action characterizing the system's next-state relation,
- v is a state function, and
- L is a formula stating the liveness conditions expected from the system.

This formula essentially describes a state machine, augmented by liveness condition, that generates the allowed behaviors of the system under specification. For the more specific classes of reactive systems, in particular the classes of reactive systems we have considered in this thesis:

- For discrete systems, a specification is a formula of the form $\mathit{Spec} \equiv \mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L_f$, where L_f is a conjunction of formula $\mathrm{WF}_v(A)$ and $\mathrm{SF}_v(A)$ where A is an action which appears as disjunct of *Next*.
- For real-time systems, a specification is a formula of the form $\mathit{RTSpec} \equiv \mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge \mathit{RTNow}(v) \wedge \mathit{RT}$ where
  - $\mathit{RTNow}(v)$ is the formula that asserts that *now* (the variable used to model real-time) is initially equal to 0 and it increases monotonically and without bound and

  - RT is a conjunction of real-time bound formulas $RTBound(A_i, v, t_i, d_i, e_i)$ where A_i is a sub-action or disjunct of $Next$, t_i, d_i and e_i is the timer, lower bound and upper bound of A_i, respectively.

- For parameterized systems, a specification is a formula of the form $parSpec \equiv Init \wedge \Box[\exists\, k \in M \,:\, Next(k)]_v \wedge \forall\, k \in M \,:\, L_f(k)$.

In our methodology, we use a class of diagrams called predicate diagrams as abstract representation of the discrete systems being considered. Assume given a specification of discrete system *Spec* and a temporal formula F, the verification of discrete systems using our diagrams can be done in two steps:

- The first step is to find a diagram that conforms *Spec*. To prove that a diagram conforms to a specification, we equip the diagram with a corresponding conformance theorem in order to produce some proof obligations. The proof is done deductively either manually by hand or by using an automatic theorem prover.

- The second step is to prove that all traces through the diagram satisfy F. In this step, we view the diagram as a finite transition system that is amenable to model checking. All predicates and actions that appear as labels of nodes or edges are then viewed as atomic propositions. Regarding predicate diagrams as finite labeled transition systems, their runs can be encoded in the input language of standard model checkers such as SPIN.

Thus, our methodology can be viewed as an integration between deductive and algorithmic verification techniques.

In Section 5.6, we have successfully proven the completeness of predicate diagram. The proof is done in four steps:

- The construction of formula automaton $\mathcal{M}^f$ which is a Muller automaton accepting exactly the behaviors satisfying F.

- The construction of specification automaton $\mathcal{M}^s$, which is a Muller automaton such that the accepting condition is defined in a way such that it exactly characterizes the fairness of *Spec*.

- The construction of product automaton $\mathcal{M}^p$, which is the product automaton of $\mathcal{M}^f$ and $\mathcal{M}^s$. Thus, the properties of $\mathcal{M}^p$ are inherited from $\mathcal{M}^f$ and $\mathcal{M}^s$.

- The last step is the translation of the product automaton into predicate diagram.

We have also shown that the concept of predicate diagrams is capable enough to handle some other classes of reactive systems such as real-time systems and parameterized systems. To verify real-time systems, we define a variant of predicate diagrams called *timed predicate diagrams* or TPDs. The idea of these diagrams is to use the components of predicate diagrams related to discrete properties and to replace the components related to the fairness conditions with some components related to real time conditions. For the components related to real-time property, we adopt the structure of timed-automata. Thus, in one direction, TPDs can be viewed as an extension of predicate diagrams. In the other direction, we may say that predicate diagrams are restricted TPDs. Particularly, when we eliminate all the components of timed predicate diagrams that are related to real-time property, then we have predicate diagrams without fairness conditions. We call such a predicate diagram the *untimed version* of a TPD. In the context of parameterized systems, we have shown that the (ordinary) predicate diagrams can still be used for proving the properties that are related to the whole processes. Whereas to prove the universal properties, i.e. the properties that are related to one single process, we define a class of diagrams called *parameterized predicate diagrams* or PPDs.

The verification of real-time systems and parameterized systems using TPDs and PPDs are similar to the verification of discrete systems using predicate diagrams.

Using the concept of abstract interpretation we have shown that our diagrams can be generated semi-automatically. We use the term "semi-automatically", since the user's intervention is still needed, in particular for defining the abstraction functions and rewriting rules. We have developed two prototype tools: PreDiaG, for the generation of predicate diagrams, and parPreDiaG, for the generation of PPDs.

Some possible tracks for future work that come to mind are listed below.

- **Hybrid systems**. Basically, hybrid systems can be viewed as the union of discrete and real-time systems. However, it is still needed to study the special characteristic of this class of systems and to investigate the extension or modification that should be done over predicate diagrams.

- **Completeness of TPDs and PPDs**. In this work, we only consider the completeness of predicate diagrams. The proof of the completeness

of TPDs and PPDs should be an interesting topic for a research. For proving the completeness of TPDs, it is indicated, that we can use the concept of timed automata and do the similar proof as we did in proving the completeness of predicate diagrams. However, this indication is still needed to be explored. Unfortunately, the proof of PPDs is still an open question.

- **Tool support**. For the practical application of our method tool support is essential. The tools we have implemented are prototypes that are still needed to be improved, in particular in the aspect of graphical user interface. We have shown that the generation of diagrams can be done incrementally. We should or may refine the diagrams resulted by our tools, until we get the desired diagrams. Thus, there is also a need to have a good graphical editor that can support the refinement of the diagrams. It is also desirable to have a translator from TLA$^+$ to MONA syntax and to integrate these tools with an existing automatic theorem prover in order to prove the proof obligations whenever needed.

Bibliography

[1] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543-1571, September 1994.

[2] Martin Abadi and Stephan Merz. On TLA as a logic. In Manfred Broy, editor, Deductive Program Design, NATO ASI series F, pages 235-272, Springer-Verlag, Berlin, 1996.

[3] M.W. Alford, J.P. Ansart, G. Hommel, L. Lamport, B. Liskov, G.P. Mullery and F.B. Schneider. Distributed Systems: Methods and tools for specification. Volume 190 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181-185, October 1985.

[5] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. Technical Report 86-727, Cornell University, Ithaca, New York, January 1986.

[6] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing 2*, pp. 117-125, 1987.

[7] Rajeev Alur. Timed automata. NATO ASI Summer School on Verification of Digital and Hybrid Systems, 1998.

[8] Rajeev Alur, C. Courcoubetis and David L. Dill. Model-checking for real-time systems. In *Proceeding of the 5th Annual Symposium on Logic in Computer Science*, pp 414-425. *IEEE Computer Society Press*, 1990.

[9] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science* 126:183-235, 1994.

[10] R. Alur and T.A. Henzinger. A really temporal logic. In *Proc. 30th IEEE Symp. Found. of Comp. Sci.*, pages 164-169, 1989.

[11] R. Alur and T.A. Henzinger. Logics and models of real time: A survey. In J.W. de Bakker, C. Huizing, W.P. de Roever and G. Rozember, editors, *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 74-106. Springer-Verlag, 1992.

[12] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, Volume 15, pp. 307-309. 1986.

[13] Kai Baukus, Yassine Lakhnech and Karsten Stahl. Verifying Universal Properties of Parameterized Networks. Technical Report TR-sT-00-4, CAU Kiel, July, 2000.

[14] Kai Baukus, Saddek Bensalem, Yassine Lakhnech and Karsten Stahl. Abstracting WS1S Systems to Verify Parameterized Networks. In *Proceeding of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, Volume 1785 of *Lecture Notes in Computer Science*, pages 188-203. Springer, 2000.

[15] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, Y. Wang and Carsten Weise. New Generation of UPPAAL. *Int. Workshop on Software Tools for Technology Transfer*. June 1998.

[16] S. Bensalem, Y. Lakhnech and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In *Conference on Computer Aided Verification (CAV-98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319-331. Springer-Verlag, 1998.

[17] S. Bensalem, et.al. An overview of SAL. In C M. Holloway, editor, *LFM 2000: 5^{th} NASA Langley Formal Methods Workshop*, pages 187-196, 2000.

[18] M. Bozzano and G. Delzanno. Beyond Parameterized Verification. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002*. Volume 2280 of *Lecture Notes in Computer Science*, pages 221-235. Springer, 2002.

[19] Anca Browne, Luca de Alfaro, Zohar Manna, Henny B. Sipma and Tomás Uribe. *Diagram-based Formalisms for the Verification of Reactive Systems*. In *CADE-13 Workshop on Visual Reasoning*, New Brunswick, NJ, July 1996.

[20] Anca Browne, Zohar Manna and Henny B. Sipma. Generalized verification diagrams. In *15th Conference in the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 484-498, December, 1995.

[21] Anca Browne, Zohar Manna and Henny B. Sipma. Modular verification diagrams. Technical report Computer Science Departement, Stanford University, 1996.

[22] R.E. Bryant. Graph-based algorithmics for boolean function manipulation. *IEEE Transactions on Computers* C-35(8):677-691.

[23] J.R. Büchi. Weak second-order arithmatic and finite automata. Z. Math. Logik Grundl. Math., 6:66-92, 1960.

[24] J.R. Büchi. On a decision method in restricted second order arithmatic. *Proceedings of the International Congress on Logic, Method and Philysophy in Science* 1960, Stanford, CA, 1962. Stanford University Press, 1-12.

[25] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang. Symbolic model checking:10^20 states and beyond. *Information and Computation* 98(2):142-170.

[26] Dominique Cansell, Dominique Méry and Stephan Merz. Predicate diagrams for the verification of reactive systems. In 2^{nd} *Intl. Conf. on Integrated Formal Methods (IFM 2000)*, vol. 1945 of *Lectures Notes in Computer Science*, Dagstuhl, Germany, November 2000. Springer-Verlag.

[27] E.M. Clarke and E.A. Emerson. Characterizing correctness properties of parallel programs using fixpoints. *International Colloquim on Automata, Languages and Programming*. Vol. 85 of *Lecture Nodes in Computer Science*, pp. 169-181, Springer-Verlag, July, 1980.

[28] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Workshop on Logic*

of Programs, Yorktown Heights, NY. Vol. 131 of *Lecture Nodes in Computer Science*, pp. 52-71, Springer-Verlag, 1981.

[29] E.M. Clarke, T. Filkorn and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis, editor. *Proceedings of the 5th Workshop on Computer-Aided Verification*. Volume 693 of *Lecture Nodes in Computer Science*, pp. 450-462. Springer, 1993.

[30] E.M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. Proceedings of the 6th annual ACM Symposium on Principles of Distributed Computing, pp. 294 - 303. Columbia, Canada, August 1987.

[31] E.M. Clarke, O. Grumberg and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.

[32] Edmund M. Clarke, Orna Grumberg and Doron A. Peled. Model Checking. The MIT Press, 1999.

[33] M.A. Colón and T.E. Uribe. Generating finite-state abstraction of reactive systems using decision procedures. In *Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer-Science*, pages 293-304. Springer-Verlag, 1998.

[34] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. Princ. of Prog. Lang.*, pp. 238-252. ACM Press, 1977.

[35] Radhia Cousot. Fondements de méthodes de preuve d'invarince et de fatalité de programmes paralléles. PhD thesis. INPL, 1985.

[36] D. Dams. Abstract interpretation and partition refinement for model checking. PhD thesis, Technical University of Eindhoven, 1996.

[37] D. Dams, R. Gerth and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 (PROCOMET)*. IFIP Transactions, North-Holland/Elsevier, 1994.

[38] D. Dams, R. Gerth and O. Grumberg. A heuristic for the automatic generation of ranking functions. In *Proceedings of Workshop on Advances in Verification*, pages 1-8. 2000.

[39] Marco Daniele, Fausto Giunchiglia and Moshe Y. Vardi. Improved Automata Generation for Linear Temporal Logic. In *Proc. 11th Intl. Conference on Computer Aided Verification*. Volume 1633 of *Lecture Notes in Computer Science*, pages 249-260. Springer, 1999.

[40] S. Das, D.L. Dill and S. Park. Experience with predicate abstractions. In *Proc. 11th Intl. Conference on Computer Aided Verification*. Volume 1633 of *Lecture Notes in Computer Science*. pages 160-171, Springer, 1999.

[41] L. de Alfaro and Zohar Manna. Temporal verification by diagram transformations. In *Proc. 8th International Conference on Computer Aided Verification*. Volume 1102 of *Lecture Notes in Computer Science*, pages 288-299. Springer, July, 1996.

[42] D. Detlefts, G. Nelson, and J. Saxe. Simplify: the ECS theorem prover. Technical report, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, November 1996.

[43] E.A. Emerson and A.P. Sistla. Symmetry and model checking. In Courcoubetis, editor. *Proceedings of 5th Workshop on Computer-Aided Verification*, pp. 463-478. June/July 1993.

[44] E.A. Emerson and K.S. Namjoshi. Automatic verification of parameterized synchronous systems. In *Proceeding of 8th Conference on Computer-Aided Verification*. Volume 1102 of Lecture Notes in Computer Science, pp. 87-98. Springer, 1996.

[45] E.A. Emerson and K.S. Namjoshi. Verification of a parameterized bus arbitration protocol. Volume 1427 of *Lecture Notes in Computer Science*, pp. 452–463. Springer,1998.

[46] Melvin Fitting. First-order logic and automated theorem proving. *Graduate Texts in Computer Science*. Springer-Verlag. 1996.

[47] Rober W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19-32, 1967.

[48] Jean H. Gallier. Logic for Computer Science: Foundation of automatic theorem proving. Harper & Row, Publisher, Inc. New York. 1986.

[49] Paul Gastin and Denis Oddoux. Fast LTL to Buchi Automata Translation. Proceedings of *13th Conference on Computer-Aided Verification*. Volume 2102 of *Lecture Notes in Computer Science*, pages 53-65. Springer, 2001.

[50] S. German and A.P. Sistla. Reasoning about systems with many processes. Journal of the ACM, Vol. 39, Number 3, July 1992.

[51] Rob Gerth, Doron Peled, Moshe Y. Vardi and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. PSTV 1995: 3-18.

[52] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of the 5th Conference on Computer-Aided Verification*. Volume 697 of *Lecture Notes in Computer Science*, pp. 438-449. Springer, 1993.

[53] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Conference on Computer Aided Verifications*. Volume 1254 of *Lecture Notes in Computer-Science*, pp. 72-83. Springer-Verlag, 1997. June 1997, Haifa, Israel.

[54] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. FME. Volume 1051 of *Lecture Notes in Computer Science*, pages 662-681. Springer, 1996.

[55] T. Henzinger, Z. Manna, and A. Pnueli. Temporal Proof Methodologies for Timed Transition Systems. *Information and Computation*, 112(2):273-337, 1994.

[56] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576-580, 1969.

[57] G. Holzmann. The SPIN model checker. *IEEE Trans. on software engineering*, 16(5):1512-1542. May 1997.

[58] Y. Kesten and A. Pnueli. Taming the Infinite: Verification of Infinite-State Reactive Systems by Finitary Means. In *Engineering Theories of Software Construction, (NATO) Science Series*, Series III: Computer and Systems Sciences, Vol. 180, pages 261-299, IOS Press 2001.

[59] Y. Kesten, Z. Manna and A. Pnueli. Verification of Clocked and Hybrid Systems. In G. Rozenberg and F.W. Vaandrager, editors, Lectures on

Embedded Systems, volume 1494 of *Lecture Notes in Computer Science*, pages 4-73. Springer-Verlag, 1998.

[60] E. Kindler. Safety and Liveness Properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, Vol. 53, pp. 268-272, 1994.

[61] N. Klarlund and A. Møller. MONA Version 1.3 UserManual. BRICS, 1998.

[62] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time Systems*, 2(4):255-299, 1990.

[63] R. Koymans and W.-P. Roever. Examples of a real-time temporal logic specifications. In B.D. Denvir, W.T. Harwood, M.I. Jackson and M.J. Wray, editors, *The analysis of concurrent systems*. Volume 207 of *Lecture Notes in Computer Science*, pages 231-252. Springer-Verlag, 1985.

[64] R. Koymans, J. Vytopyl and W.-P. de Roever. Real-time programming and asynchronous message passing. In *Proc. 2nd ACM Symp. Princ. of Dist. Comp.*, pages 187-197,1983.

[65] Fred Kröger. Temporal logic of programs. EATCS Monographs on Theoretical Computer Science, Vol. 8. Springer-Verlag. 1986.

[66] Robert P. Kurshan. Computer Aided Verification of Coordinating Processes:The automata-theoretic approach. Princeton University Presss. 1994.

[67] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):435-455, 1974.

[68] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125-143, March, 1977.

[69] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3) : 872-923, May 1994.

[70] Leslie Lamport. TLA in Pictures. *SRC Research Report 127*, Digital System Researh, California, 1994.

[71] Leslie Lamport. Introduction to TLA. *SRC Technical Node 1994-001*, Digital System Research, California. December, 1994.

[72] Leslie Lamport. Specifying concurrent systems with TLA$^+$. In *Calculation System Design*. M. Broy and R. Steinbrüggen, editors. IOS Press, Amsterdam, 1999.

[73] Leslie Lamport. Specifying Systems: The TLA+ Language and Tools or Hardware and Software Engineers. Addison-Wesley, 2002.

[74] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1), 1995.

[75] Zohar Manna, Michael Colon, Bernd Finkbeiner, Henny Sipma and Tomás Uribe. Abstraction and Modular Verification of Infinite-State Reactive Systems. In *Requirements Targeting Software and Systems Engineering (RTSE)*. Volume 1526 of *Lecture Notes in Computer-Science*, pp 273-292. Springer, 1998.

[76] Zohar Manna and Amir Pnueli. A Hierarchy of Temporal Properties. In *Proc. ACM Symposium on Principles of Distributed Computing*, 1990.

[77] Zohar Manna and Amir Pnueli. Models for reactivity. *Acta Informatica*, 30:609-678, 1993.

[78] Zohar Manna and Amir Pnueli. Clocked Transition Systems. Technical Report STAN-CS-TR-96-1566, Dept. of Computer Science, Stanford University. April, 1996.

[79] Zohar Manna and Amir Pnueli. Temporal verification diagrams. In *Proc. Intl. Symposium on Theoretical Aspects of Computer Software*. Volume 697 of *Lecture Notes in Computer Science*, pages 726-765. Springer-Verlag, 1994.

[80] Zohar Manna and Amir Pnueli. Verification of parameterized programs. In *Specification and Validation Methods (E. Borger, ed.)*, Oxford University Press, pp. 167-230, 1994.

[81] Zohar Manna and Amir Pnueli. Temporal verification of reactive systems: safety, Springer-Verlag New York, Inc., New York, NY, 1995.

[82] K.L. McMillan. Symbolic model checking:an approach to the state explosion problem. Kluwer Academic, 1993.

[83] Stephan Merz. Logic-based analysis of reactive systems: hiding, composition and abstraction. Habilitationsschrift. Institut für Informatik. Ludwig-Maximillians-Universitẗ, Munich Germany. December 2001.

[84] R. McNaughton. Testing and generating infinite sequence by a finite automaton. Inform. Contr. 9, pages 521-530, 1966.

[85] J. Misra and K.M. Chandy. Parallel program design: a foundation. Addison-Wesley Publishers, 1988.

[86] D.E. Muller. Infinite sequences and finite machines. In *Proc. 4th IEEE Symp. on Switching Circuit Theory and Logical Design*, 99:3-16, 1963.

[87] C.E. Nugraheni. Prediag: A tool for the generation of predicate diagrams. In *Proceeding of Student Research Forum, SOFSEM 2002*, pp. 35–40, November 2002.

[88] J.S. Ostroff. Formal methods for the specification and design of real-time safety critical systems. In *Journal of Systems and Software*, Vol. 18, Number 1, April 1992.

[89] J.S. Ostroff. Temporal logic of real-time systems. Advanced Software Development Series. Research Studies Press (John Wiley & Sons), Taunton, England, 1990.

[90] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455-495, July 1982.

[91] PAX Tool:Parameterized systems Abstracted and eXplored. Avaiable at `http://www.informatik.uni-`$_{\vee}$`kiel.de/~kba/pax/`.

[92] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In Dill, editor. *Proceedings of the 1994 Workshop on Computer-Aided Verification*. Volume 818 of *Lecture Notes in Computer Science*, pages 377-390. Springer-Verlag, 1994.

[93] Doron Peled. Software reliability methods. Texts in Computer Science. Springer, 2001.

[94] Amir Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium Foundation of Computer Science*, pages 46-57, IEEE Computer Society Press, 1977.

[95] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Cianzaglini and Ugo Montanari, editors, *International Symposium on Programming*. Volume 137 of *Lecture Notes in Computer Science*, pp. 337-350. Springer-Verlag, 1981.

[96] M.O. Rabin. Decidability of second-order theoried and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1-35, 1969.

[97] Fred B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Departement of Computer Science, Cornell University, Ithaca, New York, October 1987.

[98] H. Saidi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Conference on Computer-Aided Verification (CAV'99)*. Volume 1633 of *Lecture Notes in Computer-Science*, pages 443-454, Trento, Italy, 1999, Springer-Verlag.

[99] Henny B. Sipma. Diagram-based verification of discrete, reactive and hybrid systems. PhD Thesis, Dept. of Computer Science, Standford University, 1999.

[100] A.P. Sistla. On the characterization of safety and liveness properties in temporal logic. In *Proceeding of the 4th annual ACM Symposium on Principles of Distributed Computing*, pages 39-48, Minaki, Ontario, Canada, August, 1985. ACM.

[101] Fabio Somenzi and Roderick Bloem. Efficient Büchi Automata from LTL Formulae. In *the 12th Conference on Computer Aided Verification (CAV'00)*. Volume 1633 of *Lecture Notes in Computer Science*, pages 247-263. Springer Verlag, 2000.

[102] STERIA - Technolgies de l'Information, Aix-en-Provence (F). *Atelier B, Manual Utilisateur*, 1998. Version 3.5.

[103] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume II:Formal Methods and Semarntics*, pages 134-191. Elsevier Sciences Publishers B.V., 1990.

[104] TReX Examples: Fischer protocol in `http://www_verimag.imag.fr/~annichin/trex/demos/fischer.html`.

[105] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd Workshop on Computer-Aided Verification*. Volume 663 of *Lecture Notes in Computer Science*, pages 260-175. Springer-Verlag, 1992.

[106] Pierre Wolper. Constructing automata from temporal logic formulas: A tutorial. In Lectures on Formal Methods in Performance Analysis (First EEF/Euro Summer School on Trends in Computer Science). Volume 2090 of *Lecture Notes in Computer Science*, pages 261-277. Springer-Verlag, July 2001.

[107] P. Wolper and V. Lovinfose. Verifying properties of large sets of processes with network invariants. In J. Sifakis (ed), *Automatic Verification Methods for Finite State Systems*. Volume 407 of *Lecture Notes in Computer Science*, pages 68-80. Springer-Verlag, 1990.

[108] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceeding of the First Symposium on Logic in Computer Science*, pages 322-331. Cambridge, June 1986.

[109] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1-37, 1994.

[110] Sergio Yovine. Kronos: A verification tool for real-time. In *International Journal of Software Tools for Technology Transfer*, Vol. 1, Nber. 1+2, p.123-133, December 1997. Springer-Verlag Berlin Heidelberg 1997.

Appendix A

Automata generation

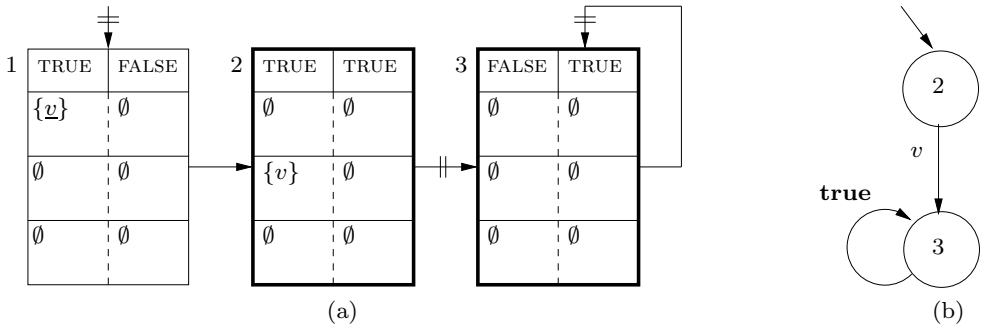

Figure A.1: Formula graph and Muller automaton for v.

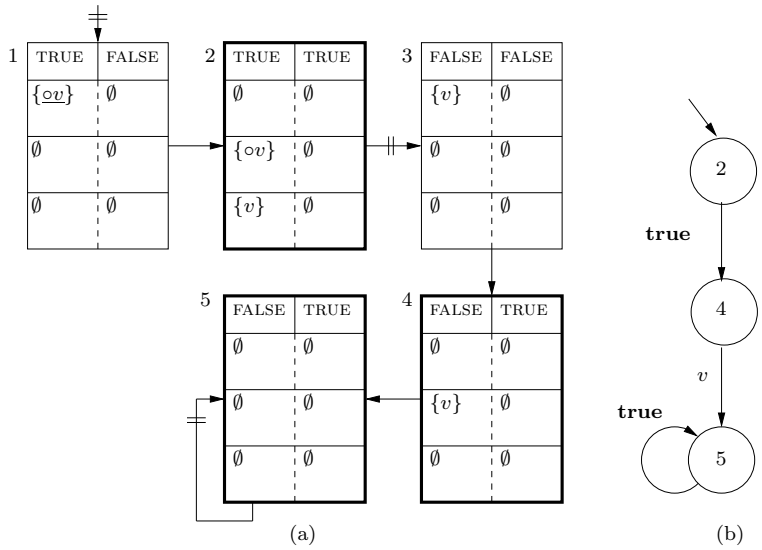

Figure A.2: Formula graph and Muller automaton for $\circ v$.

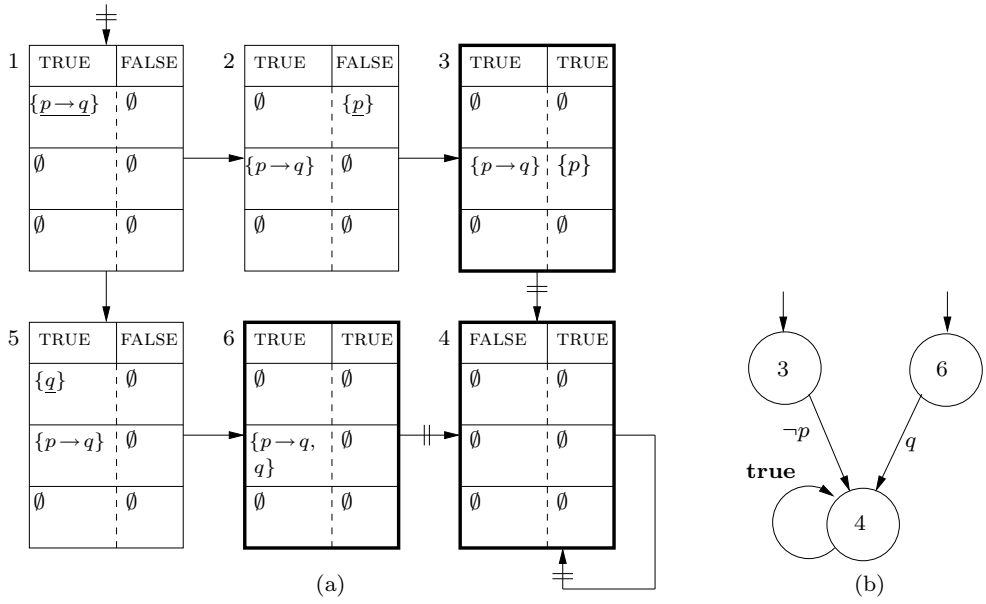

Figure A.3: Formula graph and Muller automaton for $p \rightarrow q$.

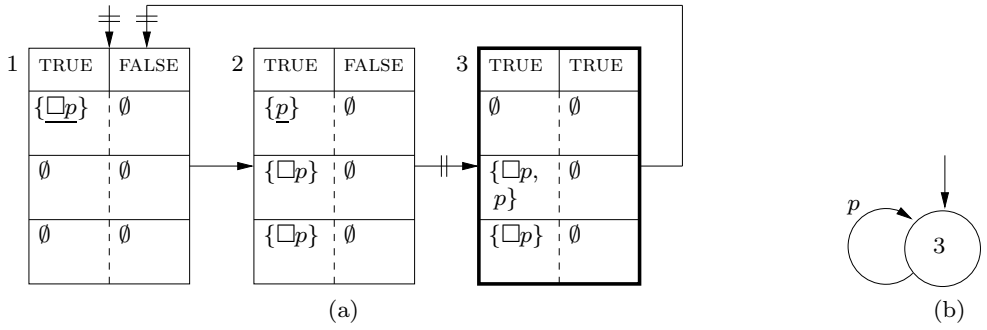

Figure A.4: Formula graph and Muller automaton for $\Box p$.

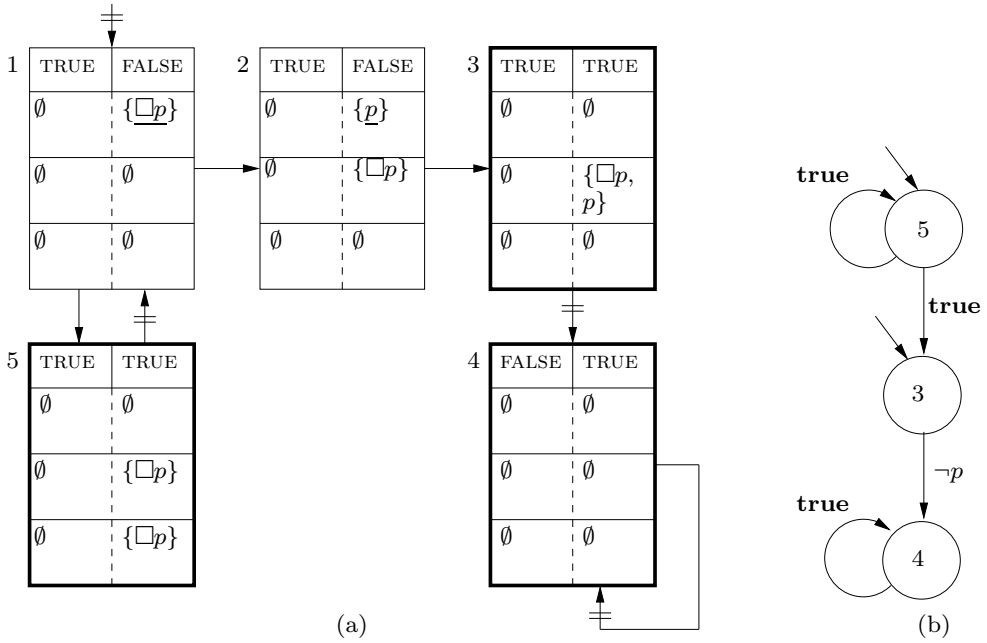

Figure A.5: Formula graph and Muller automaton for $\neg\Box p$.

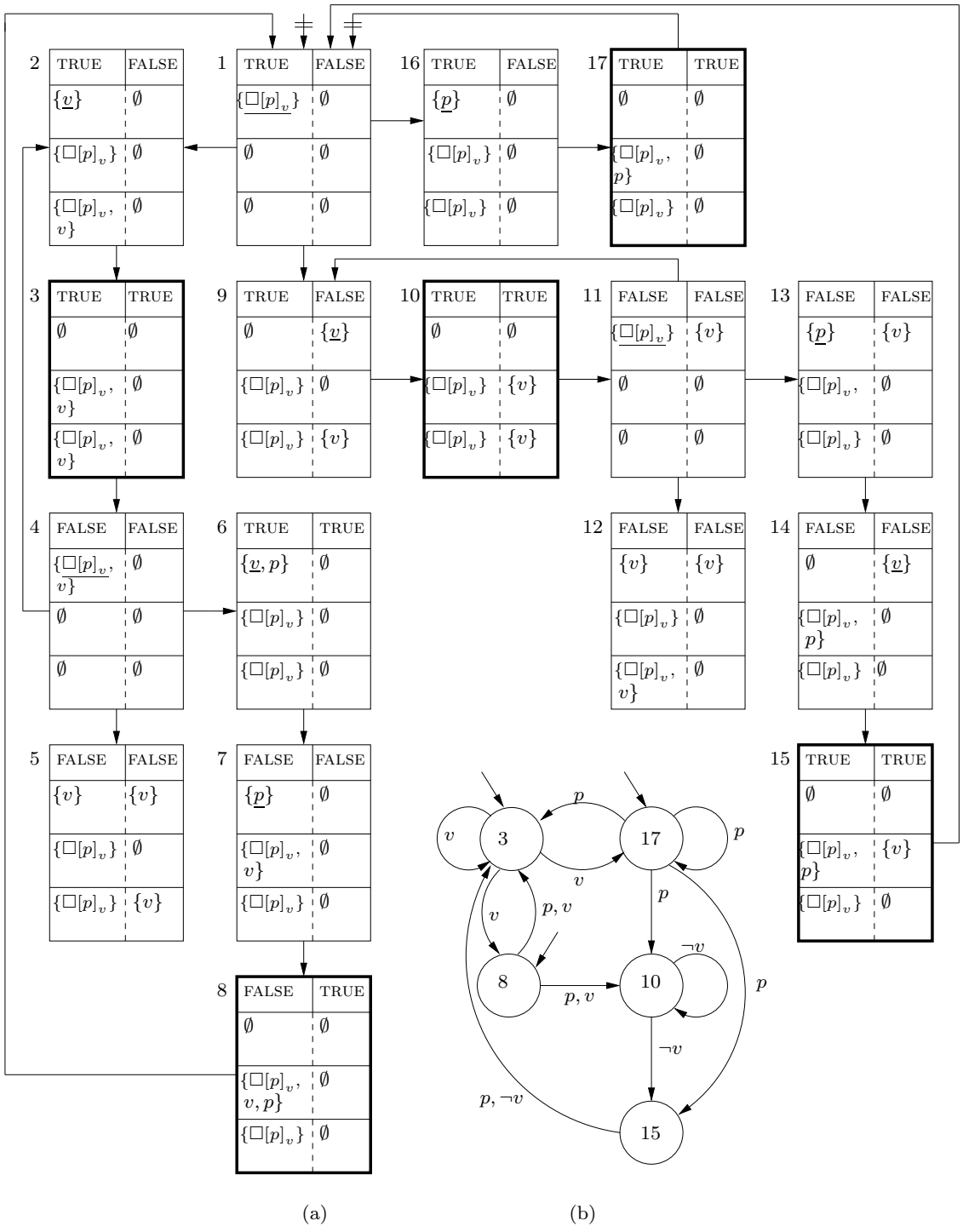

Figure A.6: Formula graph and Muller automaton for $\Box[p]_v$.

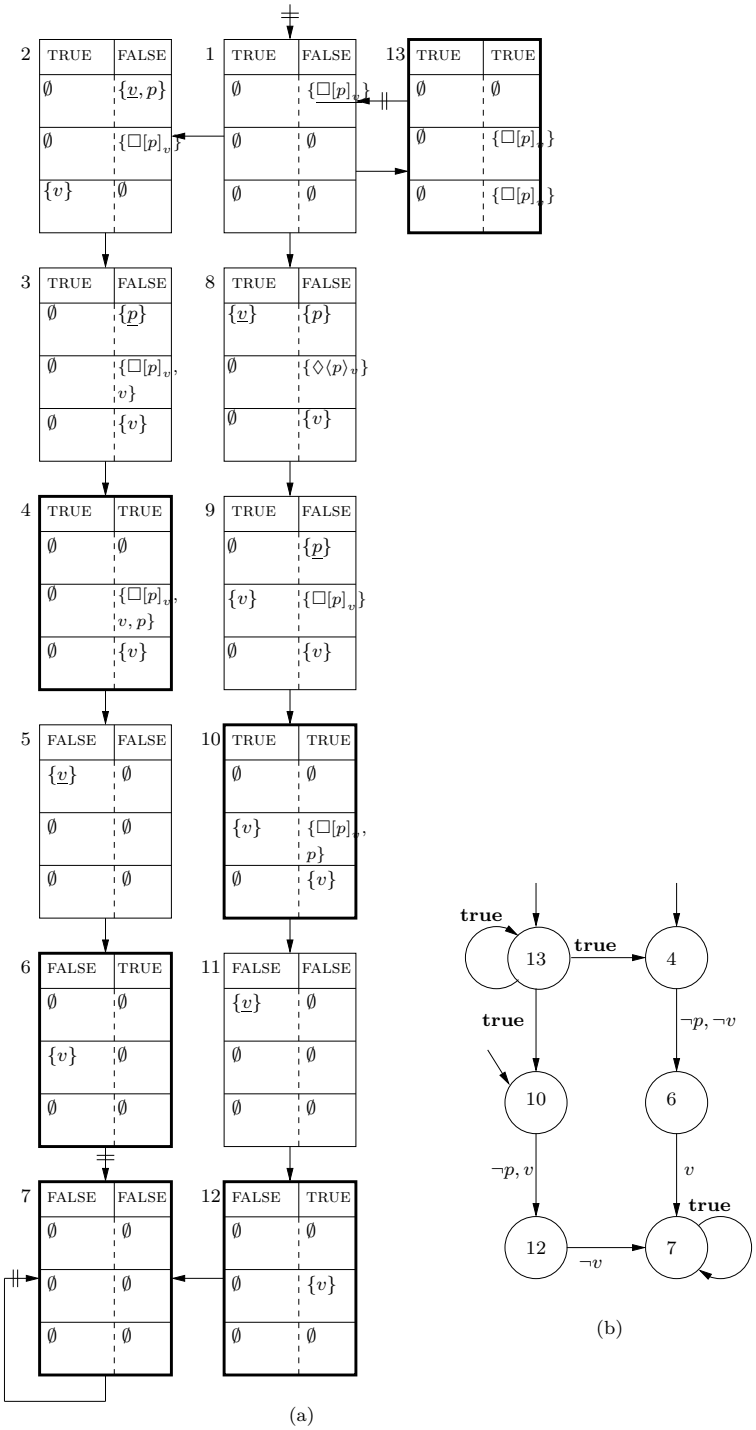

Figure A.7: Formula graph and Muller automaton for $\neg\Box[p]_v$.

Appendix B

PreDiaG

B.1 Architecture

There are four main components of PreDiaG: Front-end, Abstract states generator, Rewriting engine and Output generator. Each of these components will be briefly discussed in the sequel.

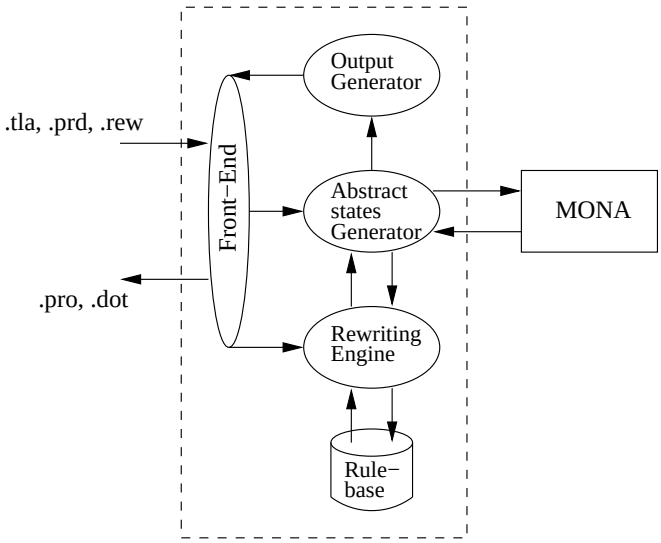

Figure B.1: Architecture of PreDiaG.

1. Front-end

   This component receives the input files from the user. It gives the information extracted from .tla and .prd to the abstract states generator, whereas the information extracted from .rew file will be given to the rewriting engine component. From the output generator it receives the

representations of the generated predicate diagrams as Promela code and as .dot file.

2. Abstract states generator

   This component receives the abstract specification and the state predicates declaration from the Front-end component. It generates the abstract states and gives the result to the output generator. We use the tool MONA [61] for the generation of the abstract states. This component gives the results of simplification process and the constraints to MONA and receives the abstract states that satisfy those formulas.

3. Rewriting engine

   This component receives a set of rewriting rules from the Front-end component and stores them in a table (rule-base). During the generation process this module receives formulas from the Abstract state generator, simplifies the formulas using the rewriting rules in rule-base and give the simplified formulas to the Abstract state generator.

4. Output generator

   This component consists of two modules: the module that produces the representation of predicate diagram as .dot file and the module that produces the representation of predicate diagram in Promela language .pro. These two modules then give the produced representations to Front-End component.

B.2 Input-Output

In order to generate the predicate diagrams, this tool needs three input files, namely: specification file (.tla), state predicate declaration file (.prd) and rewriting rules file (.rew).

The first output of this tool is the representation of predicate diagrams in Promela (.pro) and the second output is the graphical representation of predicate diagrams (.dot).

Assume we have specification $Spec \equiv Init \wedge \Box[Next]_v \wedge L$ and we want to generate the predicate diagram that conforms to $Init \wedge \Box[Next]_v$ using PreDiaG. Then the specification file should contains *Init* and every action formula which appears as disjunct in *Next*. The predicate file should contain a list of predicates in $\mathcal{P}$ and a list of constraints we will use. The rewriting rules file may contain a list of abstraction function and a list of rewriting rules.

B.3 Examples

B.3.1 AnyY problem

In the case of AnyY problem, we use the AnyY.tla, AnyY.prd and AnyY.rew as specification, predicate and rewriting rules files. The content of those files are shown in Figure B.2, Figure B.3 and Figure B.4. Figure B.5 is the graphical representation of the resulted diagram for this problem.

```
------ MODULE AnyY ------
Init == /\ x = 0
        /\ y = 0

P1 == /\ x = 0
      /\ y' = y + 1
      /\ x' = x

P2 == /\ x = 0
      /\ x' = 1
      /\ y' = y
=========================
```

Figure B.2: Specification file: AnyY.tla

B.3.2 Bakery algorithm

The input files for the Bakery algorithm are Bakery.tla, Bakery.prd and Bakery.rew which are shown in Figure B.6, B.8 and B.9, respectively. The generated predicate diagram for Bakery is given in Figure B.10.

```
(* state predicates *)
a1 == ax = 0
a2 == ax = 1
a3 == ay = zero
a4 == ay = pos

(* constraints *)
a1 <=> ~(a2)
a3 <=> ~(a4)
```

Figure B.3: Predicate file: AnyY.prd.

```
(* abstraction function *)
x = 0 => ax = 0
x' = 1 => ax' = 1
x' = x => ax' = ax
y = 0 => ay = zero
y' = y + 1 => ay' = ay + 1
y' = y => ay' = ay

(* rewrite rules *)
zero + 1 => pos
pos + 1 => pos
0 = 0 => true
0 = 1 => false
1 = 0 => false
1 = 1 => true
zero = zero => true
zero = pos => false
pos = zero => false
pos = pos => maybe
~(true) => false
~(false) => true
```

Figure B.4: Rewriting file: AnyY.rew.

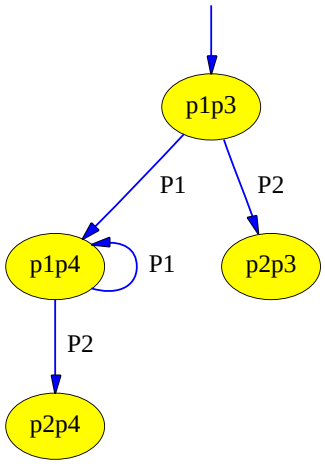

Figure B.5: Output file: AnyY.dot.

```
------ MODULE Bakery ------
Init == /\ pc1 = 0
        /\ pc2 = 0
        /\ t1 = 0
        /\ t2 = 0

NC1 == /\ pc1 = 0
       /\ pc1' = 1
       /\ pc2' = pc2
       /\ t1' = t1
       /\ t2' = t2

R1 == /\ pc1 = 1
      /\ pc1' = 2
      /\ t1' = t2 + 1
      /\ pc2' = pc2
      /\ t2' = t2

T1 == /\ pc1 = 2
      /\ t2 = 0
      /\ pc1' = 3
      /\ pc2' = pc2
      /\ t1' = t1
      /\ t2' = t2

T11 == /\ pc1 = 2
       /\ t1 <= t2
       /\ pc1' = 3
       /\ pc2' = pc2
       /\ t1' = t1
       /\ t2' = t2

C1 == /\ pc1 = 3
      /\ pc1' = 4
      /\ pc2' = pc2
      /\ t1' = t1
      /\ t2' = t2

E1 == /\ pc1 = 4
      /\ pc1' = 0
      /\ t1' = 0
      /\ pc2' = pc2
      /\ t2' = t2
```

Figure B.6: Specification file: Bakery.tla.

```
NC2 == /\ pc2 = 0
       /\ pc2' = 1
       /\ pc1' = pc1
       /\ t1' = t1
       /\ t2' = t2

R2 == /\ pc2 = 1
      /\ pc2' = 2
      /\ t2' = t1 + 1
      /\ pc1' = pc1
      /\ t1' = t1

T2 == /\ pc2 = 2
      /\ t1 = 0
      /\ pc2' = 3
      /\ pc1' = pc1
      /\ t1' = t1
      /\ t2' = t2

T21 == /\ pc2 = 2
       /\ ~(t1 <= t2)
       /\ pc2' = 3
       /\ pc1' = pc1
       /\ t1' = t1
       /\ t2' = t2

C2 == /\ pc2 = 3
      /\ pc2' = 4
      /\ pc1' = pc1
      /\ t1' = t1
      /\ t2' = t2

E2 == /\ pc2 = 4
      /\ pc2' = 0
      /\ t2' = 0
      /\ pc1' = pc1
      /\ t1' = t1
=========================
```

Figure B.7: Specification file: Bakery.tla (continued).

```
(* state predicates *)
p1 == apc1 = 0
p2 == apc1 = 1
p3 == apc1 = 2
p4 == apc1 = 3
p5 == apc1 = 4
p6 == apc2 = 0
p7 == apc2 = 1
p8 == apc2 = 2
p9 == apc2 = 3
p10 == apc2 = 4
p11 == at1 = zero
p12 == at1 = pos
p13 == at2 = zero
p14 == at2 = pos

(* constraints *)
p1 <=> ~(p2 \/ p3 \/ p4 \/ p5)
p2 <=> ~(p1 \/ p3 \/ p4 \/ p5)
p3 <=> ~(p2 \/ p1 \/ p4 \/ p5)
p4 <=> ~(p2 \/ p3 \/ p1 \/ p5)
p5 <=> ~(p2 \/ p3 \/ p4 \/ p1)
p6 <=> ~(p7 \/ p8 \/ p9 \/ p10)
p7 <=> ~(p6 \/ p8 \/ p9 \/ p10)
p8 <=> ~(p7 \/ p6 \/ p9 \/ p10)
p9 <=> ~(p7 \/ p8 \/ p6 \/ p10)
p10 <=> ~(p7 \/ p8 \/ p9 \/ p6)
p11 <=> ~(p12)
p13 <=> ~(p14)
```

Figure B.8: Predicate file: Bakery.prd.

```
(* abstraction function *)
pc1 => apc1
pc1' => apc1'
pc2 => apc2
pc2' => apc2'
t1 = 0 => at1 = zero
t2 = 0 => at2 = zero
t1' = 0 => at1' = zero
t2' = 0 => at2' = zero
t1' = t1 => at1' = at1
t2' = t2 => at2' = at2
t1' = t2 + 1 => at1' = at2 + 1
t1 <= t2 => at1 <= at2
t2' = t1 + 1 => at2' = at1 + 1
~(t1 <= t2) => ~(at1 <= at2)

(* rewrite rules *)
zero + 1 => pos
pos + 1 => pos
0 = 0 => true
0 = 1 => false
0 = 2 => false
0 = 3 => false
0 = 4 => false
1 = 0 => false
1 = 1 => true
1 = 2 => false
1 = 3 => false
1 = 4 => false
2 = 0 => false
2 = 1 => false
2 = 2 => true
2 = 3 => false
2 = 4 => false
3 = 0 => false
3 = 1 => false
3 = 2 => false
3 = 3 => true
3 = 4 => false
4 = 0 => false
4 = 1 => false
4 = 2 => false
4 = 3 => false
4 = 4 => true
zero = zero => true
zero = pos => false
pos = zero => false
pos = pos => maybe
zero <= zero => true
zero <= pos => true
pos <= zero => false
pos <= pos => maybe
```

Figure B.9: Rewriting file: Bakery.rew.

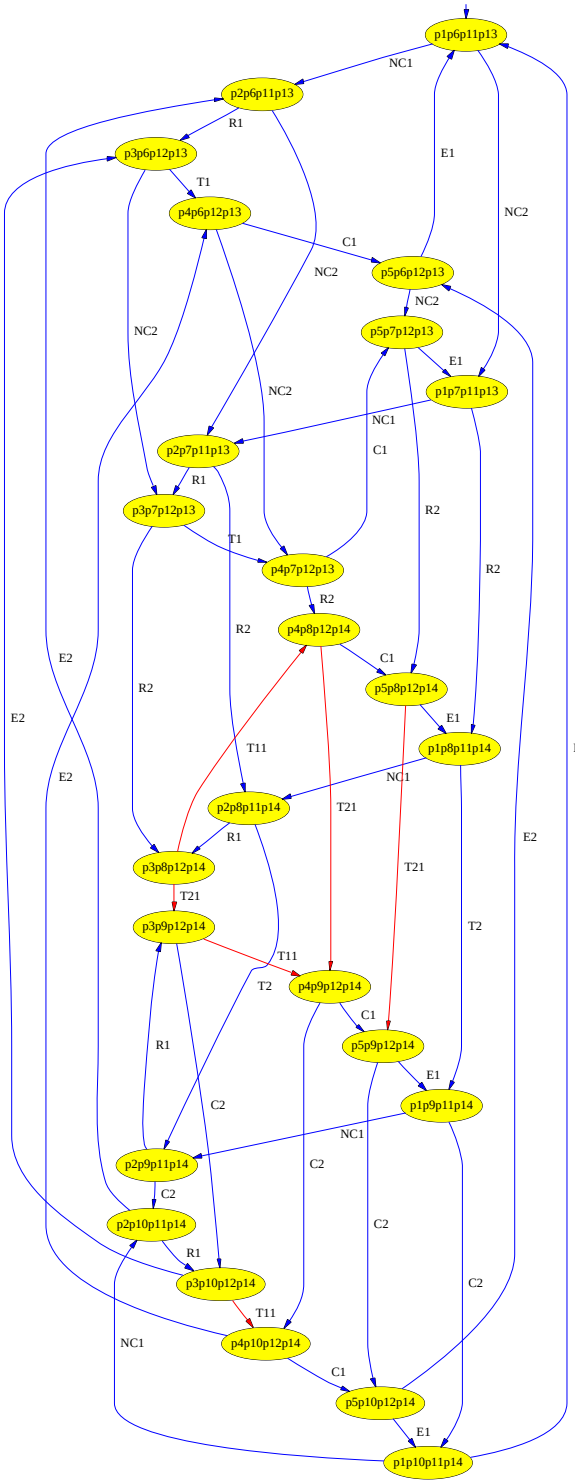

Figure B.10: Output file: Bakery.dot.

Appendix C

parPreDiaG

C.1 Architecture

There are three main components of parPreDiaG: Front-end, Abstract states generator and Output generator. The function of each component is similar to the one of PreDiaG.

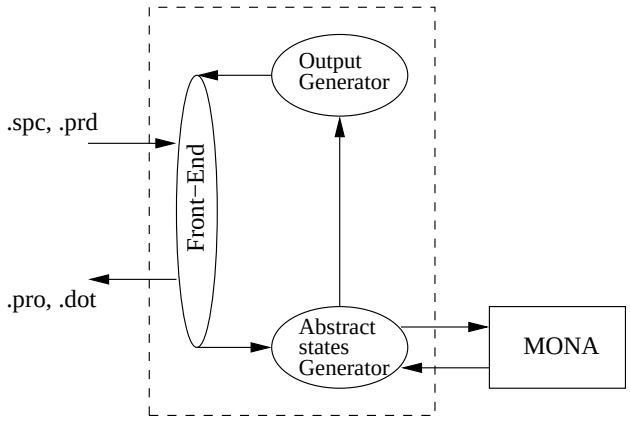

Figure C.1: Architecture of parPreDiaG.

C.2 Input and output

Two files are needed in order to generate PPDs using parPreDiaG: specification and description files. The specification file should contain a list of action formulas written in the input language of MONA and the main body of the program. The second input file, the predicate file contains a list of abstract variables, an initial condition, a list of formulas representing the quantified

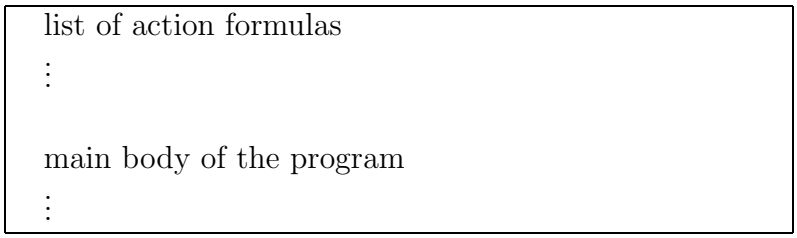

Figure C.2: The template of specification files.

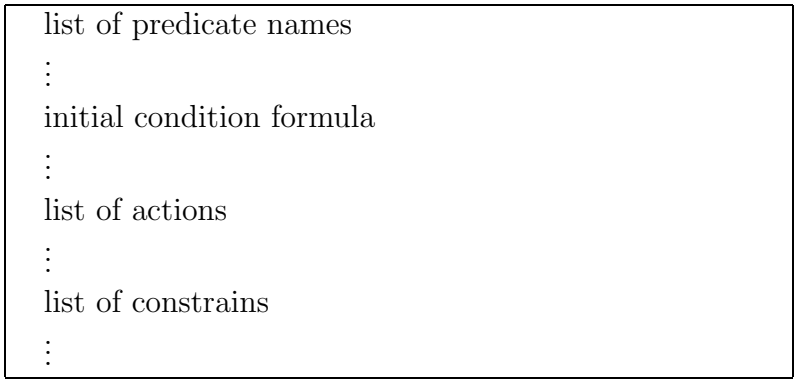

Figure C.3: Template for predicate file.

actions and a list of constraints. The templates of these files are given in Figure C.2 and C.3 in Appendix C.

The out files produced by ParPrediag are the representation of PPDs Promela (.pro) and the graphical representation of PPDs (.dot).

C.3 Example: Tickets protocol

The input files for Tickets protocol are Tickets.spc and Tickets.prd which are shown in Figure C.4 and Figure C.5.

The resulted PPD is given in Figure C.3. Notice that every edge is labeled by action name $A1..A6$. Every action name corresponds to some action declared in the predicate file. The index of every action name represents the appearance order of its corresponding quantified action in the declaration, for example $A1$ represents the action formula $Request(k, PC1, PC2, PC3, PC1', PC2', PC3')$ and $A6$ represents the formula $(ex1k : (k \sim= i \& Release(k, PC1, PC2, PC3, PC1', PC2', PC3')))$.

```
(* predicates declaration *)
p1
p2
p3
p4
p5

(* initial condition *)
i in PC1 & PC1 \ {i} ~= {} & PC2 = {} & PC3={}

(* transitions *)
Request(i, PC1, PC2, PC3, PC1', PC2', PC3')
Grant(i, PC1, PC2, PC3, PC1', PC2', PC3')
Release(i, PC1, PC2, PC3, PC1', PC2', PC3')
(ex1 k:(k~=i & Request(k, PC1, PC2, PC3, PC1', PC2', PC3')))
(ex1 k:(k~=i & Grant(k, PC1, PC2, PC3, PC1', PC2', PC3')))
(ex1 k:(k~=i & Release(k, PC1, PC2, PC3, PC1', PC2', PC3')))

(* abstractions *)
(p1 <=> i in PC1)
(p2 <=> i in PC2)
(p3 <=> i in PC3)
(p4 <=> all1 j,k: j in PC3 & k in PC3 => j=k)
(p5 <=> PC3 ~= {})
(p1' <=> i in PC1')
(p2' <=> i in PC2')
(p3' <=> i in PC3')
(p4' <=> all1 j,k: j in PC3' & j in PC3' => j=k)
(p5' <=> PC3' ~= {})
```

Figure C.4: Predicate file: Tickets.prd.

```
pred Request(var1 k, var2 PC1,PC2,PC3,PC1',PC2',PC3') =
   k in PC1 & PC1'=PC1\{k} & PC3'=PC3 & PC2'=PC2 union {k};

pred Grant(var1 k, var2 PC1,PC2,PC3,PC1',PC2',PC3') =
   k in PC2 & PC3={} & PC2'=PC2\{k} & PC3'=PC3 union {k} &
   PC1'=PC1;

pred Release(var1 k, var2 PC1,PC2,PC3,PC1',PC2',PC3') =
   k in PC3 & PC3'=PC3\{k} & PC1'=PC1 union {k} & PC2' = PC2;

ex1 i: ex2 PC1, PC2, PC3, PC1', PC2', PC3': (
true
# inserted codes
);
```

Figure C.5: Specification file: Tickets.spc.

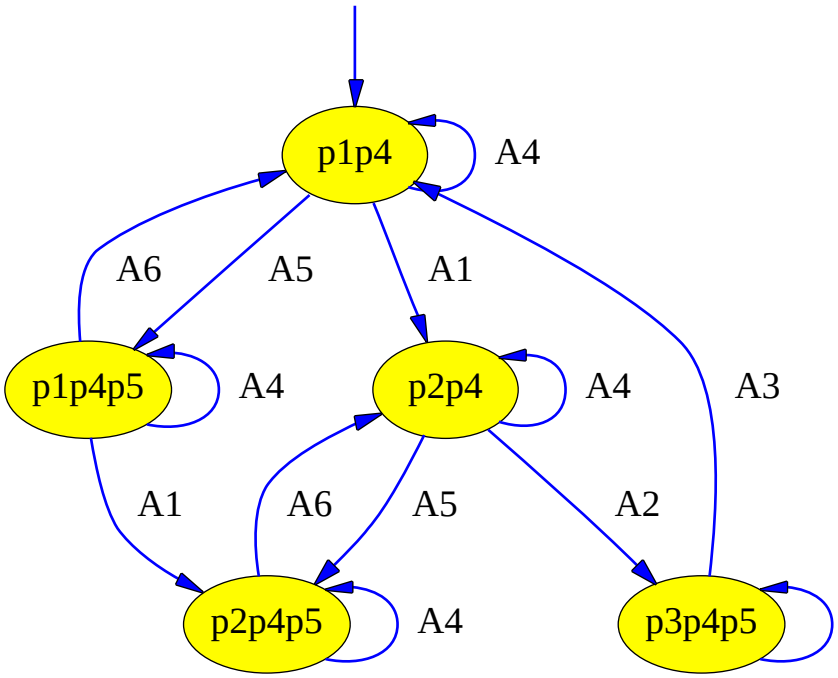

Figure C.6: Output file: Tickets.dot.

Acknowledgment

I am very grateful to my *Doktorvater* Prof. Dr. Fred Kröger for giving me the opportunity to work at the institute. Thanks for guiding me along the way, especially in the last two years. My special thanks also go to my *zweiter Betreuer* Dr. Stephan Merz for his excellent supervision. His endless constructive input have had a major impact on this thesis. It was really a privilege for me to work with them.

I am also indebted to Prof. Wirsing for the recommendations and the financial support that enabled me to attend some advanced courses related to the topic of this thesis. *Herzlichen Dank!*

A part of my tools was developed during my stay at the Department of Computering Science, Université Henri Poincaré in Nancy, France. I thank Prof. Dominique Méry, Dr. Dominique Cansell and Dr. Dennis Roegel for the assistance. *Merci!*

Financially, my work was supported by a scholarship of DAAD whose generosity is gratefully acknowledged.

I would also like to thank Dr.dr. Oerip S. Santosa MSc. for introducing me to temporal logics; Dr. A. Rusli and Rosa de Lima Ssi. MT. for understanding me every time I changed my schedule; and all the people at the Faculty of Mathematics and Natural Sciences, Universitas Katolik Parahyangan, Bandung, Indonesian, for their continuous supports. *Terima kasih!*

I would like to thank all the members of the Hans-Sachs-Ring community for the wonderful friendship. Especially, I thank Dr. Hesti Wulandari for pulling me through several times at critical moments when I was ready to give up. *Matur nuwun!*

Although far away, my mother, sister, brothers and in-laws were always there and I thank them for their constant prayer, love and encouragement.

Last, my thanks go to my husband who took over all my responsibilities (except this work). Thanks for standing by me and supporting me throughout the years. *You're the best!*

Lebenslauf

Name	**Cecilia Esti Nugraheni**
Geburtsdatum	27. November 1969
Geburtsort	Surakarta-Indonesien
Familienstand	verheiratet (seit 30. März 1999)
Staatsangehörigkeit	indonesisch
Schulausbilding	1976 - 1982 staatliche Volkschule 88, Surakarta-Indonesien 1982 - 1985 staatliche Mittelschule 1, Surakarta-Indonesien 1985 - 1988 staatliche Oberschule 1, Surakarta-Indonesien
Hochschulausbildung	1988 - 1993 Studium der Informatik am Institut Teknologi Bandung, Bandung-Indonesien (Diplom) 1995 - 1997 Studium der Informatik am Institut Teknologi Bandung, Bandung-Indonesien (Master)
Promotion	Beginn Oktober 1999 der vorliegenden Dissertation am Institut für Informatik, Lehrstuhl für Programmierung und Softwaretechnik der Ludwig-Maximilians Universität.