ABSTRACT INTERPRETATION TECHNIQUES FOR THE VERIFICATION OF TIMED SYSTEMS

Dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor in Computer Science

by

ROBERT CLARISÓ VILADROSA

PhD Program in Software
Dept. de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Barcelona
June 2005

Contents

List of Figures

List of Tables

Acknowledgements

A thesis involves lots of work and time. The result cannot be attributed to a single individual as it is enriched by many contributions from a number of people.

First, I would like to thank my PhD advisor Jordi Cortadella for his advice and motivation throughout the thesis. I am very grateful to him for many things: for introducing me to research, for the opportunities that he has provided, for all the things he has taught me and for leading by example in terms of hard work and enthusiasm.

I would also like to thank my colleagues of the research group in the Departament de Llenguatges i Sistemes Informàtics at UPC: David Bañeres, Josep Carmona, Kyller Gorgonio, Jiangtao Meng, Nilesh Modi, Albert Oliveras and Enric Rodríguez-Carbonell. It has been a pleasure working with them. They have been a great source of motivation, ideas, research tips, LaTeX tricks and programming recipes. Even when performing more mundane tasks, such as moving tables around campus, wearing red shirts with dignity or exploring Indian cuisine, they always managed to keep their wonderful sense of humor.

During my research, I have had the opportunity to share an office with many people with different backgrounds: Montserrat Civit, Elisabet Comelles, Gerard Escudero, Juan Francisco Fernández, Muntsa Padró and Francis Real. The kindness of other people made me feel at home even outside my office: Victoria Arranz, Jordi Atserias, Mauro Castillo, David Conejero, Montse Cuadros, Jesús Giménez, Enrique Romero and Lluis Villarejo. I would like to thank all of them for sharing many good moments and creating that nice and relaxed atmosphere where hard work is not so hard.

My family deserves all my gratitude for the all their love, support and patience. Thanks to my parents Ramon and Pilar, and to my brother Ramon. There are no words to express my gratitude towards my girlfriend Diana: thanks for being there, for all your love and for all that we share.

Finally, this research would not have been possible without the funding of several entities. Among them, I would like to acknowledge: the Spanish Ministry of Education and Science through the doctoral FPU grant AP2002-3862 and the CYCIT projects MAVERISH (TIC2001-2476) and GRAMMARS (TIN2004-07925), Intel Corporation, the Working Group on Asynchronous Circuit Design (ACID-WG, IST-1999-29119), and the DATE community and the EDA association for a travel grant to attend ASPDAC 2004.

Abstract

In many systems, correctness can be established simply by comparing the responses to environment events with those described in the specification. However, in some domains, correctness depends not only on *what* responses are produced, but also on *when*. The specification of such systems, called *real-time* or simply *timed systems*, should incorporate temporal information such as the delays of internal actions and external events. This concept can be generalized into *parametric timed systems*, where the specification may keep part of the temporal information as *parameters* of the system. Verifying a parametric timed system consists in determining the set of acceptable values for these parameters that ensure conformance to the specification.

In some areas, the designer of a real-time system may have control over part of its temporal characteristics. Formalizing the system as a parametric timed system can defer the choice of delays until verification is performed. Then, delay values can be selected according to the constraints discovered by verification, avoiding conservative decisions made a priori. A suitable choice of delays may improve the efficiency of the system, e.g. reducing the latency. An example of such an area of application is the design of *timed circuits*, where the behavior may depend on the delays of the gates and wires of the circuit. Techniques like gate sizing or delay insertion can be used to control these delays.

The drawback of parametric timed systems is the complexity of the verification problem, even harder than that in timed systems. Most formulations of the verification problem are undecidable. Therefore, verification procedures are either semi-decision procedures with no guarantee of termination or approximate methods which may produce inconclusive answers, e.g. *"yes"*, *"no"* or *"i don't know"*. Furthermore, most methods cannot handle problems with more than a few parameters.

This thesis addresses the verification of parametric timed systems, with a special focus on the domain of timed circuits. The proposed methods are based on the theory of *abstract interpretation*. Abstract interpretation is a general framework for the static analysis of the dynamic behavior of systems, which is used in static analysis of programs, code optimization and verification among other areas. Using safe approximation and an extrapolation technique called widening, abstract interpretation manages to analyze infinite state systems with guaranteed termination and providing conservative results. An important decision is the choice of the type of approximation, what is called the *abstract domain*. Each abstract domain offers a trade-off between precision and efficiency. In the context of parametric timed systems, there are several abstract domains which can manipulate a significant number of parameters with a reasonable precision. An example is the *convex polyhedron* abstract domain, based on linear inequality properties, e.g. $(\sum_{i=1}^{n} c_i \cdot x_i \geq k)$, where c_i and k are rational constants and x_i are numerical variables.

The contributions of this thesis belong to two areas: the verification of timed systems and the framework of abstract interpretation.

There are several approaches for the verification of timed circuits, using different families of timing constraints like metric timing or relative timing. This thesis presents a methodology based on another type of timing constraints, where gate delays can be modeled as symbols, and the timing constraints describe restrictions on the values of those symbols. The constraints discovered by this method offer several advantages with respect to previous approaches, among others, technology independence, improved precision and a simple validation procedure. The timing analysis algorithm is based on linear relation analysis with convex polyhedra. The entire verification flow has been automated: given a specification and an implementation of a timed circuit, the tool outputs the timing constraints without any need for user intervention. Experimental results show the applicability of the technique to the verification of asynchronous controllers.

Another contribution of this thesis is the description of a new abstract domain, called *octahedron*, which can be used to represent and manipulate restricted linear inequalities of the form $(\sum_{i=1}^{n} c_i \cdot x_i \geq k)$, where x_i are numerical variables, k is a rational constant and the coefficients c_i can only be $\{-1, 0, +1\}$. This family of properties captures accurately the timing constraints that arise in a timed circuit. Furthermore, this class of constraints is adequate to describe relevant properties in other static analysis problems.

This thesis characterizes the octahedron abstract domain, presents some theoretical results such as the existence of a canonical form, and describes two alternative implementations: one based on decision diagrams, and another based on bit-vectors. An advantage of these implementations is that they do not rely on the *double description method* used in convex polyhedra. As a result, octahedra achieve important memory gains with respect to convex polyhedra for the verification of timed circuits. In the decision diagram implementation, this reduction comes with a price: a large increase in the CPU time, used to minimize and reduce the decision diagram. The implementation based on bit-vectors still shows a reduction of memory while keeping a better trade-off with execution time. In terms of precision, results show that the precision achieved with octahedra is close to that of convex polyhedra in the problem under study. All these evidences support the convenience of using the octahedron abstract domain instead of convex polyhedra in those problems where it is appropiate.

Chapter 1

Introduction

Let me explain what I do here. I don't want to confuse you any more than absolutely necessary.

—Eugene Ormandy

1.1 Motivation

1.1.1 Formal Verification

In the last 20 years, the common trend shared by software and hardware industries has been an exponential increase in the complexity of designs. Even assuming very low error rates, complex designs are more prone to errors, that should be detected before the system is finally implemented. A single undetected error in a safety-critical application such as a flight-control system, a medical system or a military system can have catastrofic consequences. In some software systems, the impact of errors may be reduced by issuing patches or new versions. However, in many scenarios, such as embedded or hardware systems, correcting an error after fabrication is costly and sometimes impractical.

Error detection is typically performed through *testing* and *simulation*. However, the main problem of testing is that it cannot cover all possible scenarios. Typically, there are corner cases that are not covered by the test patterns. Quite often, errors can happen precisely in those corner cases that were not considered during the design. Clearly, there are applications for which testing and simulation is not enough.

Verification is the formal procedure that checks that the behavior of a system satisfies its specification. The benefits of a formal verification is that all possible inputs described in the specification are covered in the analysis. In this way, a system that is successfully verified can be assured to be 100% correct. However, in order to perform verification one has to deal with a huge number of states that appear when exploring the possible configurations of a system. Very often, the number of states grows very quickly with respect to the size of the design. In some cases, the state space may even be *infinite*. This substantial growth is known as the *state explosion problem*. The majority of contributions in the area of verification deal with theory, algorithms and data structures to overcome the state explosion problem.

1.1.2 Timed Circuits

In logic synthesis, most circuits are based on a *synchronous* design style. In this design style, a periodic signal called clock controls when the evolution of the state: each period corresponds to a change in the state of the circuit. An advantage of using a clock signal is that, as long as the period of the clock is long enough, the behavior of the circuit is independent from the delays of the gates and wires. The drive for continuous improvement in the field of logic synthesis has guided research to more aggressive design styles. Goals like reduced power consumption or better exploitation of concurrency have lead to relaxing the restrictions from synchronous design. In some design styles, the price to be paid for these improvements is a behavior that depends on the delays of the elements.

A circuit that relies on timing constraints to ensure its correct operation is called a *timed circuit*. Such circuits may be correct or incorrect depending on the delays of the gates, wires and environment events. The verification of a timed circuit may consist in checking its correctness, given a complete description of the delays. Another version of the problem is the discovery of the necessary timing constraints given a partial description of the delays. Several types of timing constraints have been considered, leading to different verification approaches. In any case, the verification of a timed circuit is much more complex than the verification of an untimed circuit.

This thesis will study a verification approach for timed circuits where delays do not need to be fixed in the specification: the verification of timed systems with symbolic delays. This approach is closely related to the field of verification of timed and parametric timed systems.

1.1.3 Abstract Interpretation

There are several approaches for the automatic verification of a system. For instance, *model checking* [56] is an automated technique that, given a model of a system and a property, checks whether the property holds for a given initial state of the model. *Theorem proving* [32] relies on defining the specification and the system as logical formulas in a formal logic and checking whether the implementation implies (or is logically equivalent to) the specification. *Abstract interpretation* [63] models the dynamic behavior of a system as a system of equations. The equations capture an abstraction of the state of system, which contains only the information relevant to checking the specification.

Concurrent systems can have parameters, internal variables or other elements whose value can be relevant to the verification. Furthermore, when time is taken into account, clocks, delays, response times and other temporal characteristics influence the correctness of the system. The verification process requires the study of the complex numerical properties arising among these variables.

Many verification problems that involve the discovery of numerical properties are either undecidable or have a very high computational complexity. A good candidate to solve this type of problems is the framework of abstract interpretation, where numerical properties can be studied with different *abstract domains*. Each domain encodes a family of properties with a trade-off among precision and efficiency. For instance, the domain of intervals captures the constant lower and upper bound of a variable, e.g. ($-2 \leq x \leq 7$), and provides operations like intersection or union which are linear in terms of the number of variables. However, intervals cannot represent a property like *"x is even"*. The level of precision can be selected according to the problem by choosing an adequate abstract domain. Furthermore, the framework can be extended with new abstract domains that are adapted to a specific family of properties or problems. In this thesis, a suitable domain for the verification of timed circuits will be presented.

1.2 Overview of the Contributions

The contributions of this thesis are included in two areas: the *verification of timed circuits* and the *abstract interpretation* framework. Regarding the verification of timed circuits, a new methodology that produces technology independent timing constraints has been described. With respect to abstract interpretation, a new abstract domain for the representation of numerical constraints is proposed and two alternative implementations are described. The latter set of contributions has applications to the verification of timed circuits, but also to other static analysis problems involving numerical properties.

More precisely, the contributions of this thesis are the following:

1. **A fully automatic methodology for the verification of gate-level timed circuits, based on abstract interpretation.** This methodology describes an algorithm that *automatically* generates timing constraints that guarantee the correctness of a timed circuit with respect to a safety property, e.g. conformance to the specification. In this approach, designers do not need to define lower and upper delay bounds for the elements of the circuit, something that is required in most related methods. Instead, the delays of gates and environment events are modeled using *symbols*. The output of our method is a set of *linear inequalities* involving these symbolic delays that guarantee the correctness of the timed circuit.

   A summary of the advantages provided by this methodology, which will be covered in detail within the thesis, is the following:

   - The procedure is fully automated.

- The output timing constraints produced by the algorithm are technology independent, and very easy to validate given specific constant values for the delays. Furthermore, the output constraints are meaningful, producing a valuable feedback to the designer.
- The specification can use a mixture of known and symbolic delays. The degree of parameterization can be chosen by the designer.
- The proposed method provides several benefits with respect to previous approaches based on symbolic delays, such as the support for a larger number of symbolic delays.

2. **A new numerical abstract domain for abstract interpretation called *octahedron*.** Octahedra have been specially designed to represent and manipulate the timing constraints that appear in timed circuits. An octahedron is defined as the intersection of a set of *unit* inequalities, a class of linear inequalities where coefficients are restricted to $\{-1, 0, +1\}$. The rationale behind this definition is that most timing constraints are implicitly comparing the delay of two paths within the circuit, e.g.

$$\underbrace{(\delta_1 + \cdots + \delta_i)}_{\text{delay(path}_1)} - \underbrace{(\delta_{i+1} + \cdots + \delta_n)}_{\text{delay(path}_2)} \geq k$$

In addition to the description of the abstract domain, two alternative implementations are presented: one based on decision diagrams, and another based on bit vectors. These implementations show benefits with respect to convex polyhedra.

The main strengths of this abstract domain are the following

- The octahedron abstract domain can express relational inequality properties over n variables. Few numerical abstract domains are able to express properties about an arbitrary number of variables.
- Octahedra achieve important memory gains with respect to convex polyhedra for the verification of timed circuits.
- Octahedra can also be applied to other relevant static analysis problems. There are many analysis where most interesting properties can be encoded as unit inequalities.

1.3 Organization of the Thesis

The state of the art is covered in Chapters 2 and 3. Chapter 2 introduces the theory of abstract interpretation. The verification of timed systems, and more precisely, of timed circuits is addressed in Chapter 3.

The main contributions of these thesis are presented in Chapters 4 and 5.

Chapter 4 presents a methodology to verify timed systems where delays need not be specified. Instead, each delay is represented as a symbol which is a parameter of the problem. Using abstract interpretation, a set of timing constraints that guarantee the correctness of the circuit is computed completely automatically. Each timing constraint is encoded as a linear inequality among symbolic delays. This chapter proposes the method and shows experimental results.

Chapter 5 describes an efficient representation for the timing constraints that arise in the analysis of timed systems: the octahedron abstract domain. Two alternative implementations of octahedra are described and compared to the previous approach, based on convex polyhedra. Experimental results provide an evaluation of the gains achieved with respect to the initially proposed methodology.

Then, Chapter 6 outlines further applications of the work presented in this thesis, which constitute future work in the area. The conclusions drawn from the results of this thesis are considered in Chapter 7.

Additionally, Appendix A presents the algorithms used in the decision diagram implementation of octahedra.

Chapter 2

Abstract Interpretation

There is no abstract art. You must always start with something. Later you can remove all traces of reality.

— Pablo Picasso

This chapter presents the theory behind abstract interpretation, a generic approach for the static analysis of complex systems. Special attention is devoted to the analysis of numerical properties because of their relevance in the verification of timed systems and their relationship with the contributions presented in this thesis.

2.1 Introduction

The static analysis of software and hardware systems is usually very complex. Systems with a small specification may have a very sofisticated dynamic behavior that is difficult to study. Furthermore, many problems in this context are *undecidable*, i.e. no automated procedure with guaranteed termination can solve the problem completely. In these undecidable problems, only two classes of methods can be used: *exact methods* which may not terminate for some problem instances, or *approximate methods* which do not solve the problem completely but ensure termination.

A family of methods that belong to the second category, approximate methods, is *abstract interpretation* [63]. Abstract interpretation has many applications in a wide range of areas, most of them related to the verification and optimization of software and hardware systems [60, 61]. It can be defined as a general framework for the static analysis of the dynamic behavior of complex systems. The underlying notion in abstract interpretation is that of *upper approximation*: to provide an abstraction of a complex behavior with less details. Upper approximations are conservative in the sense that they can be used to prove safety properties, e.g. “no errors in the abstraction” means “no errors in the system”. A property about a system, such as an invariant, is in some sense an abstraction: it represents all the states of the system that satisfy the property.

Intuitively, abstract interpretation defines a procedure to compute an upper approximation for a given behavior of a system. This definition guarantees (a) the termination of the procedure and (b) that the result is conservative. An important decision is the choice of the type of upper approximation, what is called the *abstract domain*. For a given problem, there are typically several abstract domains available. Each abstract domain provides a different trade-off between precision (proximity to the exact result) and efficiency.

Section 2.2 presents a quick outline of abstract interpretation with a small example. Section 2.3 provides the theoretical background on abstract intepretation. This background formalizes two important notions: approximation and extrapolation. Approximation is defined through a *Galois connection* that guarantees conservative results, while extrapolation is expressed with a *widening* operator that guarantees the termination of the analysis. Finally, Section 2.4 focuses on different abstract domains that can be used in the analysis of numerical properties.

2.2 Overview

2.2.1 Notation

Abstract interpretation can be applied to the analysis of many types of systems with different semantics. In the following, we will use a notation which is very close to software systems.

Each characteristic of the system can be described by a *variable*. A variable takes values from a given domain, e.g. a flag variable has values in $\mathbb{B}$, a counter values has values in $\mathbb{N}$. It is posible to have variables with more complex domains, e.g. a memory address, a stack of values, ... Variables that take values in $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ or $\mathbb{R}$ are called *numerical* variables.

A *state* is a complete description of the configuration of the system in a given point in time, i.e. an assignment of values to variables. The behavior of the system can be described by several *rules* defining how the state may evolve. A rule may be defined for example with a *guard*, the property required to apply the rule, and several *assignments* to the variables. Each point of the system where the state is relevant, i.e. before and after applying a rule, is called *location*. The location before a rule is called the *precondition*, while the location after a rule is called the *postcondition*.

A set of locations are labeled as the *initial locations*, which may have a set of states called *initial states*. The evolution of the state of the system, is called *execution* or *operational semantics*. An execution is described as a sequence of locations beginning with the initial location, together with the state of the system in each location.

Example 2.1 *Let us consider the following program.*

```
{$X_1$}
if (a ≤ 2)
  {$X_2$}  b := a * 2  {$X_3$}
else
  {$X_4$}  b := 8  {$X_5$}
endif
{$X_6$}
```

This program has two numerical integer variables, a and b. Each state is an assignment of values to these variables, i.e. a point in $\mathbb{Z}^2$. The rules defining the behavior of the system are the statements of the program. There is a location X_i in all the points where the state of the system is relevant: before and after each statement or guard. In this example, the goal of the analysis is the discovery of a superset of the possible values of a and b in each location.

2.2.2 Overall Strategy

The analysis of a system often consists in the enumeration of all its possible executions. An exact enumeration is difficult, because the set of reachable states may be very large or even infinite. Instead of an exact analysis, in abstract interpretation the set of states in each location is abstracted as an *upper approximation* of the exact set, i.e. a superset. An upper approximation may be more efficient to manipulate than the exact set of values.

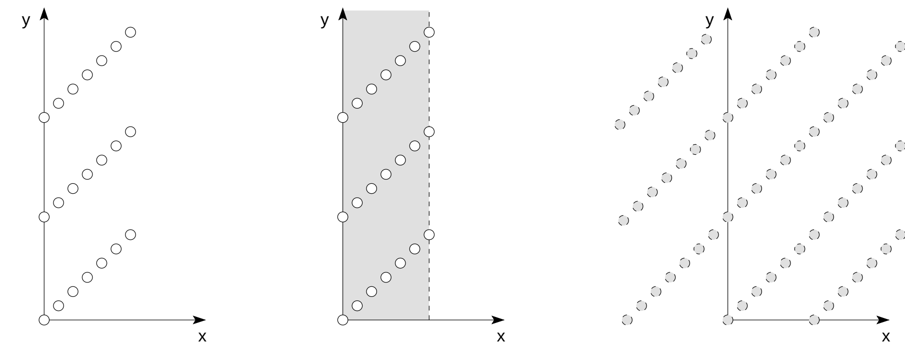
Example 2.2 *Let us consider the postcondition of the following assignment.*

```
x := y % 7
```

It is possible to represent the values of variable x *in several ways. For instance, the values of* x *can be approximated using the* interval *abstract domain [62] as* $x \in [0, 6]$. *Another option is the linear congruence equality abstract domain [86], where the postcondition is approximated as* $(x - y \equiv 0 \mod 7)$. *These representations are more compact than an explicit enumeration of the values of* x*, and each one provides a trade-off between accuracy and efficiency. Figure 2.1 shows both approximations graphically. The highlighted areas contain all the states described by each approximation. Note that, while both methods describe upper approximations of the exact postcondition, each one contains additional states that do not belong to the postcondition.*

The relationship between the precondition and the postcondition of each rule can be expressed in terms of the abstraction. To achieve this goal, each guard and assignment is rewritten as a transformation in the abstract domain that preserves the “upper approximation”.

Example 2.3 *Let us recall the program from the Example 2.1. A possible system of fixpoint equations that approximates the program in the interval abstract domain is the following:*



Exact postcondition

```
(0,0), (1,1), ...
(0,7), (1,8), ...
```

Intervals

```
{ x ∈ [0,6]}
```

Linear congruence equalities

```
{ x - y ≡ 0 mod 7 }
```

Figure 2.1: Approximating the postcondition of the assignment `x := y % 7`.

```
{$X_1$}
if ($a \leq 2$)
    {$X_2$}  $b := a * 2$  {$X_3$}
else
    {$X_4$}  $b := 8$  {$X_5$}
endif
{$X_6$}
```

$$
\begin{aligned}
X_1 &= \{\, a \in [-\infty, +\infty],\ b \in [-\infty, +\infty] \,\} \\
X_2 &= X_1 \,\cap\, \{\, a \in [-\infty, 2] \,\} \\
X_3 &= X_2\,[\, b \leftarrow a * 2 \,] \\
X_4 &= X_1 \,\cap\, \{\, a \in [3, +\infty] \,\} \\
X_5 &= X_4\,[\, b \leftarrow 8 \,] \\
X_6 &= X_3 \,\cup\, X_5
\end{aligned}
$$

The system of equations describes the program in the abstract domain of intervals. Each operation has a translation in the interval semantics. For instance, conjunction and disjunction in tests are approximated using the intersection and union of intervals. Assignments are approximated by establishing the lower and upper bound of the expression being assigned. Finally, the convergence of execution paths, e.g. at the end of an **if-then-else** *statement is approximated by the union of intervals. The solution to this system of equations can be computed by setting each non-initial location to the empty set of states and applying the equations iteratively until a fixpoint is reached, i.e. further applications of the equations do not change the set of states in any location. This strategy, called* increasing chain, *computes the following solution:*

Location	***Variable*** a	***Variable*** b	***Location***	***Variable*** a	***Variable*** b
X_1	$[-\infty, +\infty]$	$[-\infty, +\infty]$	X_4	$[3, +\infty]$,	$[-\infty, +\infty]$
X_2	$[-\infty, 2]$	$[-\infty, +\infty]$	X_5	$[3, +\infty]$,	$[8, 8]$
X_3	$[-\infty, 2]$,	$[-\infty, 4]$	X_6	$[-\infty, +\infty]$	$[-\infty, 8]$

Several aspects of the abstract interpretation framework have not been fully covered by this small example. First, the resolution of the system of equations may be more complex when there are loops. A special operator called *widening* must be used in order to guarantee the termination of the algorithm that solves the system of equations. Also, the system of equations may be written in different ways, e.g. *backward* instead of forward equations. Finally, there are many different abstract domains, which may be much more complex than the one presented in this example. All these issues will be covered in detail in the remaining of this chapter.

2.3 Formal definition

2.3.1 General Framework

There are several elements to be defined in the abstract interpretation framework: approximation, resolution of fixpoints and extrapolation.

The concept of *approximation* defines the relationship between the abstractions and the states of the system. In abstract interpretation, approximation is defined as a *Galois connection*. It is this notion what allows abstract interpretation analysis to be *conservative*.

The behavior of the system will be modeled as a set of fixpoint equations. Several strategies can be used to define and solve this system of equations, e.g. forward or backward, increasing or decreasing, different orders of evaluation, ... This approach is the “engine” behind all abstract interpretation analysis.

Extrapolation is required to ensure the termination of the fixpoint analysis in cyclic systems. Two special operators called *widening* and *narrowing* are used to extrapolate the cyclic behavior. Without them, the termination of abstract interpreation methods would not be guaranteeed.

2.3.2 Galois Connection

The states of a system can be considered as elements of a *concrete* domain C. On the other hand, the approximate values that are obtained by our analysis are elements of an *abstract* domain A. Approximation is defined by a pair of functions, the *abstraction* function ($\alpha : 2^C \rightarrow A$) and the *concretization* function ($\gamma : A \rightarrow 2^C$) which define the path from sets of concrete values to an abstract value and vice versa. This pair of functions should ensure *safety*, i.e. that the abstract value is always an *overapproximation* of the concrete values that it represents. This notion can be formalized as a Galois connection.

Definition 2.1 (Galois connection [65]) *Let* $(C, \leq)$ *and* $(A, \sqsubseteq)$ *be partially ordered domains, called the* concrete *domain and the* abstract *domain respectively. A pair of functions* $\alpha : 2^C \rightarrow A$ *and* $\gamma : A \rightarrow 2^C$ *is a* Galois connection *if and only if the following holds:*

$$\forall X \subseteq C,\ y \in A :\ \ (\alpha(X) \sqsubseteq y)\ \Leftrightarrow\ (X \leq \gamma(y))$$

In this case, α *is called the* abstraction *function and* γ *is called the* concretization *function.*

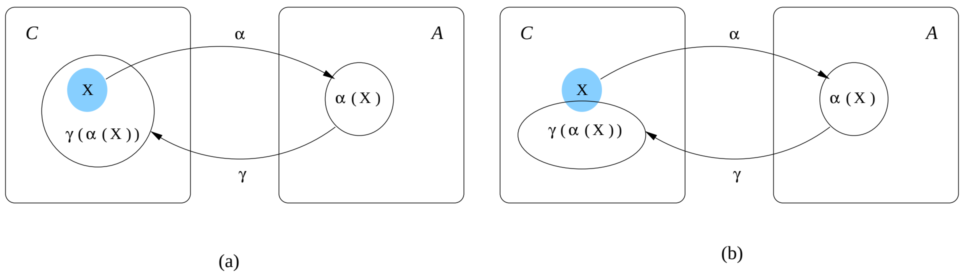


Figure 2.2: (a) A Galois connection $\langle \alpha, \gamma \rangle$. (b) A pair of functions $\langle \alpha, \gamma \rangle$ which is not a Galois connection.

Example 2.4 *Let us consider the abstraction (α) and concretization (γ) functions used in the interval abstract domain [62]. This domain is used to approximate the values of a numerical variable. Let us assume that the variable takes integer values.*

The concrete domain C is the numerical domain $\mathbb{Z}$. The abstract domain A consists of intervals, pairs of the form $[l, u]$ where l is the lower bound and u the upper bound, i.e. $(\mathbb{Z} \cup -\infty) \times (\mathbb{Z} \cup +\infty)$. There is also an empty interval, noted as $\perp$. Intuitively, each interval abstracts all the values between the lower and upper bound. The abstraction and concretization functions can be defined as:

$$\begin{array}{lrcl}
\textbf{\textit{Abstraction}} & \alpha(\emptyset) & = & \perp \\
 & \alpha(X) & = & [\min X, \max X] \\
\textbf{\textit{Concretization}} & \gamma(\perp) & = & \emptyset \\
 & \gamma([l,u]) & = & \{x \in \mathbb{Z} \mid l \leq x \leq u\}
\end{array}$$

The properties stated in the definition of a Galois connection can be stated informally as "$\alpha(X)$ is the most precise approximation of X" and "$\gamma(y)$ is the set of elements from C that can be soundly approximated as y". Any approximation satisfying these properties may lose precision when moving from one domain to the other, but it will preserve safety. Formally, this property is stated as $\forall X \subseteq C : X \subseteq \gamma(\alpha(X))$.

Example 2.5 *Figure 2.2 describes two pairs of functions. The pair depicted in (a) satisfies the definition of a Galois connection. Moving from the concrete domain to the abstract domain loses some precision, as some states not included in X are represented by the abstraction. But in any case, it is safe as all the values from the original X are faithfully represented in the abstraction. To sum up, $X \neq \gamma(\alpha(X))$ shows approximation while $X \subseteq \gamma(\alpha(X))$ shows safety.*

On the other hand, the pair shown in (b) is not a Galois connection, because $\gamma(\alpha(X))$ is not an overapproximation of X. An analysis that relies on these pairs of functions will not be conservative, e.g. it may fail to compute all the reachable states of a system.

2.3.3 Resolution of Fixpoints

The execution of the system can be rewritten as a system of equations defined in the abstract domain. Abstract intepretation describes a procedure that solves the system of equations iteratively until a fixpoint is reached [63]. This procedure can be customized according to different criteria, which will be presented in this section. Table 2.1 summarizes these criteria.

Direction of the Equations

The concept of forward and backward analysis will be familiar to the readers with a background on compilers (data-flow analysis). The system of equations can be written in two directions, *forward* and *backward*. Forward equations express the postcondition of a rule in terms of the precondition. A forward analysis starts from the initial locations and computes the reachable states iteratively. On the other hand, backward equations describe the precondition of a rule in terms of the postcondition. This analysis starts from the final locations and attempts to build a path towards the initial locations.

Example 2.6 *Let us consider a rule consisting on the following assignment:*

$$\mathtt{x} := \mathtt{x} + 1$$

Let us denote the values of x *in the precondition as* $Pre(\mathtt{x})$ *and in the postcondition as* $Post(\mathtt{x})$. *In the interval abstract domain, the forward equation and backward equations for this rule will be:*

$$\begin{array}{ll} \textit{Forward} & Post(\mathtt{x}) := Pre(\mathtt{x}) + [1,1] \\ \textit{Backward} & Pre(\mathtt{x}) := Post(\mathtt{x}) - [1,1] \end{array}$$

Forward and backward equations may be very different. For example, consider the forward and backward equations of the following assignment:

$$\mathtt{x} := 3$$

$$\begin{array}{ll} \textit{Forward} & Post(\mathtt{x}) := [3,3] \\ \textit{Backward} & Pre(\mathtt{x}) := [-\infty,+\infty] \end{array}$$

In the backward equation, the assigment to variable x *erases all the information about* x *available in the postcondition. This type of assignment is called non-invertible.*

Both systems of equations are somewhat dual, but in some cases they do not yield the same amount of information, as it is shown in Example 2.6. Intuitively, backward analysis might be more efficient in problems where a small set of final states is known a priori and the set of states reachable from the initial set is very large. An example can be a complex system where a set of errors to be avoided in known in advance, and the problem consists on checking if the errors are reachable from the initial states [60].

Target Solution

When two execution paths of the system converge, e.g. after a conditional statement in a program, the analysis might want to compute a property which covers both executions (*maximal* or *join* solution) or the property which is satisfied by both paths simultaneously (*minimal* or *meet* solution). Intuitively, maximal solutions compute a property which is a disjunction of the properties satisfied in all paths, e.g. the set of possible values of a variable. On the other hand, minimal solutions compute a property which is a conjunction of the properties satisfied in all paths, e.g. a program terminates only if *all* execution paths terminate.

Example 2.7 *Let us consider a property used in compiler optimization. An assignment is called* dead *if the value defined by the assignment is not read before being overwritten by another assignment. For instance,*

```
a := x
b := y
if ( $c \neq 17$ )
   a := 14 + b
   b := 7
else
   a := 2
   b := 9
endif
```

The assignment (a := x) is dead, because its value is always overwritten before being read. However, the assignment (b := y) is not dead, because its value is read in the statement (a := 14 + b). From this example, it is clear that an assignment is dead if it is overwritten in all *paths of execution after the assignment. An analysis to discover this type of property requires a minimal (meet) solution.*

Direction of the Chains

The system of equations, whether they are forward or backward, will be evaluated in the same way. For the purpose of our presentation, we will assume a system of forward equations.

The evaluation of a system of forward equations starts from the initial locations. Equations are applied iteratively until the fixpoint is reached. But there are two possible ways in which the fixpoint can be reached, a *increasing* chain and a *decreasing* chain.

An increasing chain initializes all locations except the initial ones with the empty set of states. Each iteration will try to increase the set of reachable states in each location. During the iterative resolution of the equations, the set of states in a location forms an increasing chain:

$$Solution^0 \sqsubseteq Solution^1 \sqsubseteq \ldots \sqsubseteq Solution^k = Solution^{k+1}$$

where $Solution^i$ is the abstract value assigned to a given location in the iterarion i. The definition of decreasing chain is the dual. Decreasing chains assume that *all* states are reachable in each location, except the initial ones. Each iteration will try to decrease the set of unreachable states in each location, forming the decreasing chain defined as:

$$Solution^0 \sqsupseteq Solution^1 \sqsupseteq \ldots \sqsupseteq Solution^k = Solution^{k+1}$$

Again, both types of analysis may provide different results, and each one can be used to complement the other. For example, an increasing chain can be used to compute the reachable states, followed by a decreasing chain to remove unreachable states from the previous result. This combination may be more precise than each of the analysis separately.

Example 2.8 *Let us consider the following program with a single integer variable i, which we assume is initialized to zero:*

```
{$X_1$}
while {$X_2$} ($i \le 2$)
  {$X_3$} i := i + 1 {$X_4$}
endwhile
{$X_5$}
```

$$\begin{aligned} X_1 &= [0,0] \\ X_2 &= X_1 \cup X_4 \\ X_3 &= X_2 \cap [-\infty,2] \\ X_4 &= X_3\,[\,i \leftarrow i+1] \\ X_5 &= X_2 \cap [3,+\infty] \end{aligned}$$

On the right, there is a system of forward equations that describes the program in the interval abstract domain. The goal in this example is the study of the possible values of the variable i in each location $X_{1\ldots4}$. The computation that leads to the fixpoint can be performed using the increasing or the decreasing chain as follows:

	Increasing chain					***Decreasing chain***		
	0	*1*	*2*	*3*	*4*	*0*	*1*	*2*
X_1	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$
X_2	$\emptyset$	$[0,0]$	$[0,1]$	$[0,2]$	$[0,3]$	$[-\infty,+\infty]$	$[-\infty,+\infty]$	$[-\infty,3]$
X_3	$\emptyset$	$[0,0]$	$[0,1]$	$[0,2]$	$[0,2]$	$[-\infty,+\infty]$	$[-\infty,2]$	$[-\infty,2]$
X_4	$\emptyset$	$[1,1]$	$[1,2]$	$[1,3]$	$[1,3]$	$[-\infty,+\infty]$	$[-\infty,3]$	$[-\infty,3]$
X_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$[3,3]$	$[-\infty,+\infty]$	$[3,+\infty]$	$[3,3]$

In the first iteration (iteration 0), the initial location is set to the initial value $[0,0]$. Other locations are set to the empty interval in the increasing chain, and the universe interval in the decreasing chain. Each iteration applies the equations sequentially, starting from X_1 until X_4. The abstract value given to a location X_j in iteration i, noted as X_j^i, is computed using the equation for X_j: the location on the right-hand side of the equation are replaced by the abstract value known in the previous iteration, e.g. X_k^{i-1}. After several iterations, the system of equations converges. Note that each approach obtains a different solution, and also that each approach requires a different number of iterations to reach the solution.

The computation shown in this example is simplified with respect to the real algorithm to clarify the presentation. Section 2.3.3 will consider efficient orders of evaluation of the equations, while Section 2.3.4 will focus on termination in the presence of iterative behavior such as loops.

Order of Evaluation

The solution of a system of fixpoint equations can be achieved by evaluating the equations iteratively until the system converges. A good order of evaluation can accelerate the convergence. The best order of evaluation depends on the characteristics of the system being studied.

For example, in simple structured functions without goto/break/continue statements and without function calls, a very efficient evaluation strategy can be defined: evaluate the statements of the program in sequence. Whenever a condition **if-then-else** is reached, evaluate one branch of the conditional and then evaluate the other one before continuing. Finally, whenever a loop is found, evaluate the loop body until it converges before evaluating the following statements. In more complex systems, such as programs with recursive functions, multi-threaded programs or programs with **goto** statements, establishing good evaluation orders is more complex.

Several methods for choosing good evaluation strategies have been discussed in the literature [30,44,64]. The basis behind these methods is the concept of *topological order* in directed acyclic graphs: an ordering of nodes in the graph such that each node appears before each of its descendants. In the context of abstract interpretation, this ordering is good because it propagates the effects of a statement to its successors naturally. The evaluation orderings for abstract interpretation attempt to find topological-like orderings for the systems of equations, which are possibly cyclic.

2.3.4 Extrapolation: Widening and Narrowing

Solving the system of equations consists in applying the equations iteratively until reaching a fixpoint. The fixpoint may be reached through an increasing chain or a decreasing chain, according to the strategy. However, there is a problem that may lead to non-termination: the abstract domain may have infinite chains. For example, an infinite increasing chain is an infinite sequence S of abstract values such that:

$$S_0 \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq \ldots \sqsubseteq S_n \sqsubseteq \ldots \quad \text{where } \forall i : S_i \neq S_{i+1}$$

In the analysis of a system, this type of infinite chains can appear when there is iterative behavior, e.g. a cycle in an automaton or a loop in a program. Each iteration of this loop may transform the abstract states in such a way that it does not stabilize.

EQUATIONS	**Forward equations**	**Backward Equations**
Meaning	$Post^{i+1} = f(Pre^i)$	$Pre^{i+1} = f(Post^i)$
Origin	Initial states	Final states

CHAIN	**Increasing chain**	**Decreasing chain**
Meaning	"Start from the empty set, then add reachable states"	"Start from the full set, then remove unreachable states"
Initial state	$Solution^0 = \alpha(\emptyset)$	$Solution^0 = \alpha(2^C)$
Invariant	$Solution^i \subseteq Solution^{i+1}$	$Solution^{i+1} \subseteq Solution^i$
Extrapolation	Widening (∇)	Narrowing ($\triangle$)

GOAL	**Maximal solution**	**Minimal solution**
Meaning	"Property holds in some path"	"Property holds in all paths"
Junction	$\cup$	$\cap$

Table 2.1: Criteria to classify abstract interpretations

Example 2.9 *Let us consider the analysis of the following program using the interval abstract domain. Initially, the value of variable i is known to be between 0 and 1.*

```
{X_1}
while {X_2} (i > 0)
  {X_3} i := i + 1 {X_4}
endwhile
{X_5}
```

$$\begin{aligned} X_1 &= [0,1] \\ X_2 &= X_1 \cup X_4 \\ X_3 &= X_2 \cap [1,+\infty] \\ X_4 &= X_3\,[\,i \leftarrow i+1] \\ X_5 &= X_2 \cap [-\infty,0] \end{aligned}$$

On the right, the system of equations that computes the forward maximal equation is described. Using the increasing chain strategy, the following solutions are computed iteratively:

	0	*1*	*2*	*3*	...	*k*	...
X_1	$[0,1]$	$[0,1]$	$[0,1]$	$[0,1]$	...	$[0,1]$	...
X_2	$\emptyset$	$[0,1]$	$[0,2]$	$[0,3]$	...	$[0,k]$	...
X_3	$\emptyset$	$[1,1]$	$[1,2]$	$[1,3]$	...	$[1,k]$	...
X_4	$\emptyset$	$[2,2]$	$[2,3]$	$[2,3]$	...	$[2,k+1]$	...
X_5	$\emptyset$	$[0,0]$	$[0,0]$	$[0,0]$	...	$[0,0]$	...

It is clear that the set of states for the locations X_2, X_3 and X_4 will not converge, so the analysis will not terminate.

Abstract interpretation introduces an extrapolation operator in order to guarantee termination. This extrapolation operator is called *widening* (∇) for increasing chains and *narrowing* ($\triangle$) for decreasing chains. Intuitively, this operator studies the changes in the state after one iteration of the loop/cycle and attempts to extrapolate the behavior of the loop from this information, assuming that the same transformation may occur an infinite number of times. A side effect of the extrapolation is that widening accelerates the convergence for systems of equations that require a large number of iterations to be solved, e.g. the program

$$\textbf{for } (i = 0; i < 10^6; i++) \; \{ \; stmt; \; \}$$

Definition 2.2 *Widening operator [65] A* widening operator ∇ *is a function* $\nabla : A \times A \to A$ *such that*

1. $\forall x, y \in A : x \cup y \sqsubseteq x \nabla y$
2. *for all increasing chains* $x^0 \sqsubseteq x^1 \sqsubseteq \ldots$*, the increasing chain defined as* $y^0 = x^0, \ldots, y^{i+1} = y^i \nabla x^{i+1}$ *is not strictly increasing.*

By property (2) of the definition, the widening operator can only be applied a finite number of times before reaching convergence. Therefore, widening can guarantee the convergence of a system of equations. The definition of the narrowing operator is equivalent, replacing the increasing chain by a decreasing chain.

Example 2.10 *Let us reconsider the analysis of the program from Example 2.9, this time using a widening operator. The widening operator for the interval abstract domain can be defined as:*

$$\begin{array}{rcl} [a,b] \; \nabla \; [c,d] & = & [e,f] \; \textit{where} \\ e & = & (\; c < a \; ? \; -\infty \; : \; a \;) \\ f & = & (\; d > b \; ? \; +\infty \; : \; b \;) \end{array}$$

The system of equations using the widening operator and the abstract values computed in each location are the following:

$$\begin{array}{rcl} X_1 & = & [0,1] \\ X_2 & = & X_2 \; \nabla \, (X_1 \; \cup \; X_4 \;) \\ X_3 & = & X_2 \; \cap \; [1,+\infty] \\ X_4 & = & X_3 \, [\, i \leftarrow i+1] \\ X_5 & = & X_2 \; \cap \; [-\infty, 0] \end{array}$$

	0	*1*	*2*	*3*
X_1	$[0,1]$	$[0,1]$	$[0,1]$	$[0,1]$
X_2	$\emptyset$	$[0,1]$	$[0,+\infty]$	$[0,+\infty]$
X_3	$\emptyset$	$[1,1]$	$[1,+\infty]$	$[1,+\infty]$
X_4	$\emptyset$	$[2,2]$	$[2,+\infty]$	$[2,+\infty]$
X_5	$\emptyset$	$[0,0]$	$[0,0]$	$[0,0]$

Notice that this time the analysis terminates after only 3 iterations. The crucial point of this computation is the location X_2 *in the second iteration. This location is computed as:*

$$\begin{array}{rcl} X_2^2 & = & X_2^1 \; \nabla \, (X_1^1 \; \cup \; X_4^1 \;) \\ & = & [0,1] \; \nabla \; ([0,1] \; \cup \; [2,2]) \\ & = & [0,1] \; \nabla \; [0,2] \\ & = & [0,+\infty] \end{array}$$

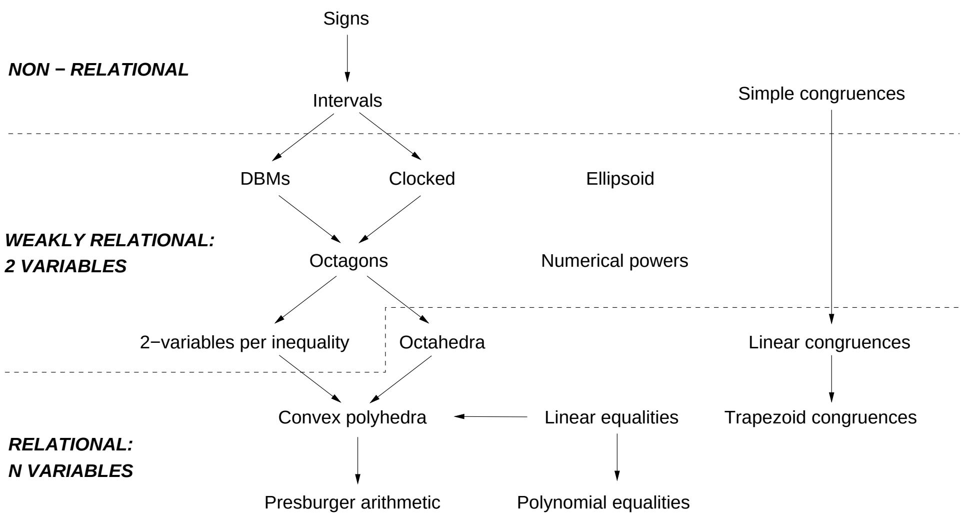


Figure 2.3: Non-exhaustive hierarchy of numerical abstract domains. An arrow between two domains $X \to Y$ denotes that X is a subset of Y, i.e. Y is more expressive.

Notice that the widening operator has detected that each iteration of the loop may increase the value of `i`*, and extrapolates that it could happen an infinite number of times. This extrapolation may be too conservative, but it ensures the termination of the analysis.*

Widening operators may cause an important loss of precision in the analysis of a system. There are several strategies to reduce this loss of precision. The first strategy is the *delayed widening* [59]. This strategy uses the union operator $\cup$ instead of the widening ∇ for a constant number of iterations. If the fixpoint is not reached during these iterations, then the operator is switched to the widening. Another strategy is the *widening with thresholds* [62, 89]: define a finite set of thresholds that represent candidate properties that might hold after the widening. Instead of extrapolating the bounds up to infinity directly, the widening extrapolates up to the next threshold. When no more thresholds are available, the extrapolation proceeds up to infinity. In the context of a specific abstract domain, convex polyhedra, there is also additional work on more precise widening operators [16].

2.4 Numerical abstract domains

2.4.1 Description of a Numerical Abstract Domain

A very active field of study in abstract interpretation is the definition of efficient abstract domains. Each problem requires discovering a different family of properties about a system, e.g. worst-case execution time, values of variables, . . . It is possible to design customized abstract domains which are well-suited to discover

	Abstraction	Citation	Properties
Inequalities	Signs	[63]	$x \in \pm$
	Intervals	[62]	$0 \leq x \leq 3$
	DBMs	[72, 113]	$x - y \leq 4$
	Octagons	[114]	$x + y \leq 2$
	2-vars per inequality	[144]	$3x + 2y \leq 4$
	Octahedra	Chapter 5	$x + y - z \leq 7$
	Convex polyhedra	[66]	$2x + 9y + 3z \leq 6$
	Presburger arithmetic	[38]	$\forall x : (x + 1 \leq 5) \vee (x \geq 1)$
Equalities	Linear equalities	[101]	$3x + 6y - 2z = 14$
	Polynomial equalities	[117, 137, 141]	$4x^2 + 2xy + 5y^3 = 7$
Congruences	Simple congruence	[85]	$x \equiv 2 \mod 11$
	Linear congruence	[86]	$4x - 2y + 3z \equiv 4 \mod 7$
	Trapezohedral congruence	[109]	$7x - y + 2z \in [2, 3] \mod 17$

Table 2.2: Description of several numerical abstract domains.

the necessary properties in a given problem. The design of this abstract domain must take into account the trade-off between precision and efficiency.

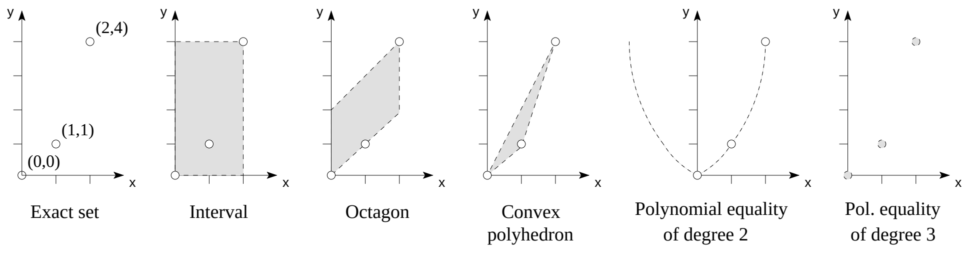
As abstract interpretation has been applied in many different areas and problems, there are many abstract domains in the literature. For instance, there are abstract domains to represent properties as complex as type information, the contents of a cache memory [1] or *alias* information [57, 69, 128], i.e. the addresses stored in the pointers of a program. However, a field where abstract interpretation is specially successful is the *analysis of numerical properties*. Numerical properties have been studied in a large group of problems such as the verification of real-time systems [71, 91], the verification of software programs [26, 29, 66, 114], or the study of data-dependences in array accesses [85, 86, 109].

The description of a numerical abstract domain consists of a Galois connection, a data structure and the abstract operators required to approximate the behavior of the program. Some examples of the required abstract operators are intersection, union, widening, and assignments.

The abstraction and concretization function that form the Galois connection can be characterized intuitively as follows. The concrete values are the states of the system. Each state of a system with n variables over the numerical domain D (which can be $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{R}$) can be seen as a point in D^n. A set of states S is a subset of D^n, i.e. $S \subseteq D^n$. On the other hand, the abstract domain is a family of properties $P : D^n \rightarrow \{true, false\}$, e.g. intervals. Each abstract value is a specific property P from the family F, e.g. $x \in [-1, 4]$. The meaning of the abstraction (α) and concretization (γ) functions is therefore:

$$\begin{array}{llcl}
\textbf{Abstraction} & \alpha(\emptyset) & = & false \\
 & \alpha(X) & = & \text{"strongest property } P \in F \text{ satisfied by } X\text{"} \\
\textbf{Concretization} & \gamma(false) & = & \emptyset \\
 & \gamma(P) & = & \{x \in D^n \mid P(x) = true\}
\end{array}$$

The ordering relation $\leq$ used in the concrete domain is typically the inclusion ($\subseteq$), e.g. $(X \leq Y) \leftrightarrow (X \subseteq Y)$. Meanwhile, the semantics of the ordering relation in



Interval	Octagon	Convex polyhedron	Polynomial equalities of degree ≤ 2	Polynomial equalities of degree ≤ 3
$0 \leq x \leq 2$	$0 \leq x \leq 2$	$y - x \geq 0$	$y = x^2$	$y = x^2$
$0 \leq y \leq 4$	$0 \leq y - x \leq 2$	$x - 2y \geq 0$		$x \cdot (x-1) \cdot (x-2) = 0$
		$3x - y \leq 2$		

Figure 2.4: Approximating a set of values (top left) with several abstract domains. With only two variables, octahedra are equivalent to octagons, and convex polyhedra are equivalent to the two-variable per inequality domain.

the abstract domain ($\sqsubseteq$) is usually based on the implication, $(P \sqsubseteq Q) \leftrightarrow (P \rightarrow Q)$.

The definition of the data structures and the algorithms will be detailed in the following sections. The presentation will emphasize previous work on the design of numerical abstract domains based on inequality properties, in order to motivate the contribution presented in Chapter 5: a numerical abstract domain that captures properties of the form $(\pm x_1 \pm \ldots \pm x_n \leq k)$, i.e. inequalities on the sum sum and difference of an arbitrary number of variables.

2.4.2 Classification of Numerical Abstract Domains

Figure 2.3 shows a non-exhaustive classification of numerical abstract domains. We have chosen three criteria to classify the differences among the abstract domains. These criteria are the following:

- The *family of properties* encoded in the abstract domain, e.g. congruences ($x \equiv 2 \; mod$ 9), equalities ($x = 7$) or inequalities ($x \leq 9$).
- The *number of variables* that appear in each property, e.g. at most one variable ($x \in [7, 8]$), at most two variables ($x - y \leq 6$) or any number of variables ($3x - 2y + z \leq 0$). Domains which only encode properties about a variable are called *non-relational*. Other domains that can encode properties about any number of variables are called *relational*, while domains that can only manipulate properties about a subset of variables at a time are called *weakly relational*.
- The *type of expressions* that appear in the properties, e.g. only differences ($x - y = 2$), sums and differences ($x + y = 6$), linear expressions ($2x -$

$3y = 4$), polynomial expressions ($3x^2 - 2y = 4$) or exponential expressions ($x = 2^y$).

Each abstract domain has a different degree of expressiveness. Some abstract domains are more expressive than others, e.g. a property stated in the domain A can always be expressed in the domain B without loss of precision. This relationship is depicted in Figure 2.3 producing a hierarchy. Domains which appear higher in the hierarchy are less precise but potentially more efficient. Figure 2.4 depicts graphically several abstract domains of this hierarchy. The shaded areas correspond to the states represented by the abstract value. Notice that some abstractions are strictly more precise than others.

Inequalities, Equalities and Congruences

Inequality abstract domains are based on computing bounds on the values of variables. Due to efficiency reasons, most of these domains compute envelopes which are *convex*, i.e. any segment between a pair of internal points will also be contained by the envelope. As the union of convex objects is not convex in general, there is some degree of approximation in the union operation. The information provided by inequality domains has been used to remove out-of-bound checks in arrays and detect out-of-bound checks at compile-time; to detect arithmetic overflow or underflow; to detect the outcome of loops or conditional statements at compile-time; and also to detect arithmetic run-time errors such as "division by zero".

Equality based abstract domains keep track of (and combine) the explicit information that appears in the assignments and guards of the system. A trivial example of application of this information is the *constant propagation* used in code optimization, i.e. detect when the value of a variable is constant in a point of the program. However, more sofisticated analysis can produce invariants which can be used to verify complex algorithms [117, 137].

Congruence abstractions provide information based on congruence relations. There are several types of analysis, each providing a different level of granularity. Information on congruence relations is used mainly for the analysis of memory accesses and data dependencies. For example, congruence relations can be use to decide whether two instructions are accessing the same position in an array. Then, a compiler can use this information for scheduling, loop parallelization or reordering of loops.

Non-Relational vs. Relational Domains

Regarding the number of variables appearing in each property, the different categories reflect clearly the trade-off between precision and efficiency. Non-relational domains can only describe properties about individual variables, i.e. for each variable a subset of D is characterized while the relationship with other variables is ignored. For example, a non-relational domain may describe the upper and lower bound of the variable, or the value modulo a fixed constant. Notice that these abstract domains cannot express that the value of variable depends on the value of another variable. For instance, they cannot represent the fact that a variable x always has the same value as variable y. Conversely, relational domains can describe relations between several variables, e.g. x is greater than y.

The advantage of non-relational domains is their low computational complexity. In general, their complexity is $O(n)$, linear in the number of variables in the worst case. On the other hand, relational domains have a higher complexity because they consider the relationships between variables. An important factor in this complexity is *the number of variables* that can appear in each property relating several variables. Domains that handle relations between pairs of variables may potentially find $O(n^2)$ such properties, while handling triples of variables may discover $O(n^3)$ properties,and so on. It follows that domains where all the variables can appear inside a property are likely to have exponential complexity. Thus, several abstract domains limit the properties being represented to pairs of variables, in order to achieve a good polynomial complexity. This domains are called weakly relational.

Types of Expressions

In relational abstract domains, the type of expressions that are supported reflects again the trade-off between precision and efficiency. Limiting the complexity of the expressions can improve the efficiency of the analysis.

Allowing only differences of variables or sums and differences reduces the number of coefficients that may appear in each variable. This sets an *upper bound on the number of possible expressions*, something that may help in the definition and implementation of the operations. On the other hand, linear expressions are also popular due the frequency in which they appear and the availability of efficient algorithms to deal with linear properties, e.g. linear programming. More complex expressions such as polynomials are very expressive, but the mathematic algorithms dealing with these properties quickly become prohibitive because of their complexity.

Other abstract domains attempt to compute a very specialized family of properties which is relevant in a concrete class of problems or systems. Some examples of these custom abstract domains are the *clocked* domain [26], the *ellipsoid* domain [26] or the *numerical powers* domain [110].

2.4.3 Intervals

Intervals are a representation for constraints on the upper or lower bound of a single variable, e.g. $\texttt{x} \in [2, 7]$. Interval analysis is very popular due to its simplicity and efficiency: an interval abstract domain for n variables requires at most $O(n)$ memory, and all operations require at most $O(n)$ time. Efficient implementations of this abstract domain may share information among different locations to reduce the complexity below $0(n)$ [26].

Another strength of interval analysis is the possibility to find precise overapproximations of complex operations such as divisions, products, modulos, ... For example, the following approximations can be used:

$$\begin{array}{rcl} [2,3] * [5,11] & \subseteq & [10,33] \\ \text{[3,10]}\ \mathrm{div}\ [7,14] & \subseteq & [0,1] \\ \text{[1,2]}\ \mathrm{mod}\ [3,4] & \subseteq & [1,2] \end{array}$$

However, intervals cannot represent non-trivial symbolic relations. For example, if $x \in [0,2]$ and $y \in [0,2]$, x and y *can* be equal, but we do not know if they are. On the other hand, sometimes trivial symbolic relations can be inferred from the upper and lower bounds: if $x \in [5,7]$ and $y \in [2,3]$ we know that $x \geq y$ for any value of x and y.

2.4.4 Difference Bound Matrices (DBMs)

The *Difference Bound Matrix* (DBM) is a data structure proposed for the study of timed systems [72]. However, it can also be used as an abstract domain [113]. A DBM stores a conjunction of lower and upper bounds on variables and the difference between pairs of variables. The implementation of a DBM is a two-dimensional matrix M where each cell $M[i,j] = k$ represents the constraint $(x_i - x_j \leq k)$. The variable x_0 encodes the constant 0, so the lower and upper bound of a variable $(k_1 \leq x_i \leq k_2)$ are represented as $(x_0 - x_i \leq -k_1)$ and $(x_i - x_0 \leq k_2)$ respectively.

Example 2.11 *A constraint of the form* $(x_1 \leq 7) \wedge (x_2 \geq 3) \wedge (x_1 - x_2 \leq 4)$ *can be encoded as the following DBM:*

	$\mathbf{x_0}$	$\mathbf{x_1}$	$\mathbf{x_2}$
$\mathbf{x_0}$	0	∞	-3
$\mathbf{x_1}$	7	0	4
$\mathbf{x_2}$	∞	∞	0

Notice that the diagonal of the matrix is full of zeros as $(x_i - x_i \leq 0)$. *Also, there are several unknown bounds in the matrix, that are encoded as* $(x_i - x_j \leq \infty)$.

DBMs have a canonical form which can be obtained using Floyd's all-pair shortest path algorithm in $O(n^3)$ time for a DBM with n variables, or $O(n^2)$ if the canonical form is computed incrementally. Regarding other operations, the emptiness test is $O(n^3)$ while the intersection, the union, the widening or the inclusion test are $O(n^2)$. For example, the intersection of two DBMs M_1 and M_2 is simply another DBM where each cell contains the minimum from the two operands, i.e. $M[i,j] = \min(M_1[i,j], M_2[i,j])$.

The memory required by a DBM is at most $O(n^2)$. Furthermore, it is possible to provide a *minimal constraint* representation [104] where redundant constraints are removed, e.g. constraints with a coefficient ∞. This representation can achieve important memory savings with a small time overhead.

Even though this abstract domain is very efficient, it is not very precise. It can represent some symbolic properties, e.g. $(x = y)$, even if the value of the variables is unknown. However, there is a loss of precision because there is no way to express constraints on the sum of variables.

2.4.5 Octagons

The domain of *octagons* [114] is an extension of the DBM abstract domain. In addition to the DBM constraints, octagons can encode bounds on the sum of pairs of variables, i.e. $(k_1 \leq x \leq k_2)$ and $(k_1 \leq x - y \leq k_2)$ and $(k_1 \leq x + y \leq k_2)$. The name “octagon” is chosen because in a system with two variables, octagons can represent at most eight constraints.

Given a set of n variables, the implementation of an octagon is a DBM-like matrix with $2n$ variables. Each original variable x_i is divided into two variables, x_i^+ and x_i^-, encoding the values of $(+x)$ and $(-x)$ respectively. In this context, some constraints appear in two cells of the matrix, e.g. $(x_1 + x_2 \leq 8)$ can be expressed as $M[x_1^+, x_2^-] = 8$ or $M[x_2^+, x_1^-] = 8$.

Example 2.12 *A constraint like* $(x_1 \leq 7) \wedge (x_2 \geq 3) \wedge (x_1 - x_2 \leq 4) \wedge (x_1 + x_2 \leq 8)$ *can be encoded as the following octagon:*

	$\mathbf{x_1^+}$	$\mathbf{x_1^-}$	$\mathbf{x_2^+}$	$\mathbf{x_2^-}$
$\mathbf{x_1^+}$	0	14	4	8
$\mathbf{x_1^-}$	∞	0	∞	∞
$\mathbf{x_2^+}$	∞	8	0	∞
$\mathbf{x_2^-}$	∞	4	-6	0

Again, the diagonal of the matrix is full of zeros. Also, notice the redundancy that appears because some constraints are encoded twice.

Regarding complexity, the operations on octagons have the same asymptotic complexity as in DBMs. The canonical form of octagons is computable in $O(n^3)$ time, or $O(n^2)$ if it is computed incrementally. Operations are again between $O(n^3)$ for the test of inclusion and $O(n^2)$ for other operations. The memory usage has grown from $(n+1)^2$ to $4n^2$, but a clever encoding can avoid storing redundant cells. The minimal constraint representation used in DBMs can also be used to reduce the memory required to store an octagon.

Regarding precision, there is a loss of precision whenever there is an assignment of the form $(\mathsf{x} := K * \mathsf{y})$ or $(\mathsf{x} := \mathsf{y} \pm \mathsf{z})$. Also, it is not possible to represent symbolic constraints involving three or more variables. These problems are inherent to the family of constraints that can be represented with this abstraction.

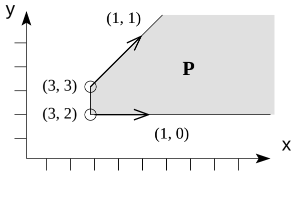
2.4.6 Convex Polyhedra

A *linear inequality* is an expression of the form $(\sum_i c_i \cdot x_i \leq k)$, where c_i and k are constants in $\mathbb{Q}$, e.g $(3x + 2y - z \leq 7)$. Convex polyhedra [66, 91] are an efficient representation for systems of linear inequalities. This abstract domain is very popular due to the ability to express powerful constraints. However, this precision comes with a very high complexity overhead.

Convex polyhedra can be represented as the set of solutions of a conjunction of *linear inequalities* with rational coefficients. Let P be a polyhedron over $\mathbb{Q}^n$, then it can be represented as the solution to the system of m inequalities $P = \{X | AX \geq B\}$ where $A \in \mathbb{Q}^{m \times n}$ and $B \in \mathbb{Q}^n$. Convex polyhedra can also be represented in a *polar* representation, called the *system of generators*, as a linear combination of a set of vertices V (points) and a set of rays R (vectors). The fact that there are two representations is important, because several of the operations for convex polyhedra are computed very efficiently when the proper representation of polyhedra is available. Figure 2.5 shows an example of a convex polyhedron and its double description.

There is a procedure that translates from one representation into the other. This procedure was described in [45], and further improved in [79, 152]. Given a system of c constraints over $\mathbb{Q}^n$, the computation of the dual representation requires $O(c^{\lfloor \frac{n}{2} \rfloor})$ time. The size of the representation can also grow exponentially with this translation, e.g. an hypercube in n-dimensions is defined by $2n$ constraints but it has 2^n vertices. The worst case is achieved by *cyclic polytopes*, which have up to $O(c^{\lfloor \frac{n}{2} \rfloor})$ vertices with a system of inequalities with c constraints.

As any operation on convex polyhedra may require a conversion from one representation to the other, all operations have a cost which is exponential with respect to the number of variables n, both in terms of time and memory. Although there is no known way of avoiding the exponential complexity, several strategies can be used to improve the efficiency of the double-description method. A usual technique is *lazy conversion*, a strategy performed in some libraries such as in [17, 99] that attempts to minimize the number of conversions between representations. In lazy conversion, a dual representation is not computed unless it is required to perform an operation. Therefore, a sequence of operations that require the same representation does not need to perform any conversion at all. Another approach, called *cartesian factoring* [90], consists in partitioning the set of variables V of the polyhedra into disjoint subsets of variables $V_1, \ldots, V_k$, such that there are no constraints among variables of different subsets. A polyhedron P is encoded as a set of polyhedra



System of generators

$$P = \{\lambda_1 \cdot (3,3) + \lambda_2 \cdot (3,2) + \mu_1 \cdot (1,1) + \mu_2 \cdot (1,0) \mid \\ \lambda_1 \geq 0, \lambda_2 \geq 0, \mu_1 \geq 0, \mu_2 \geq 0, \lambda_1 + \lambda_2 = 1\}$$

System of constraints

$$P = \{(x,y) \mid (y \geq 2) \wedge (x \geq 3) \wedge (x - y \geq 0)\}$$

Figure 2.5: An example of a convex polyhedron (shaded area) and its double description

$P_1, \dots, P_k$, where each P_i is a convex polyhedra over the variables in V_i. For instance, in a convex polyhedron $(x + 2y \leq 3) \wedge (3z - 4t \leq 5)$, the sets $\{x, y\}$ and $\{z, t\}$ are unrelated. Therefore, the original polyhedra over the variables $\{x, y, z, t\}$ can be encoded by the pair of polyhedra $(x + 2y \leq 3)$ over $\{x, y\}$ and $(z + 4t \leq 5)$ over $\{z, t\}$. This approach can reduce the number of variables in each polyhedron, therefore reducing the impact of the exponential complexity. However, its effectiveness depends on the existence of unrelated sets of variables.

The set of operations on convex polyhedra that are required for abstract interpretation are the following:

- **Test for inclusion** ($P \subseteq Q$): Inclusion is an exact operation. P is included in Q only if the generators of P satisfy the constraints of Q, that is, $\forall v \in V : Av \geq B$ and $\forall r \in R : Ar \geq 0$.

- **Union** ($P \cup Q$)): The union of convex polyhedra is not necessarily convex, and therefore an upper approximation is used. This approximation is called *convex hull*, the least convex polyhedron that includes P and Q. $P \cup Q$ is defined as the polyhedron with a system of generators that is the union of those in P and Q.

- **Intersection** ($P \cap Q$): The intersection of two convex polyhedra is necessarily convex. $P \cap Q$ can be defined as the polyhedron with a system of linear inequalities that contains all the inequalities in P and Q.

- **Widening** ($P \nabla Q$): Widening is the approximate operator used to guarantee termination in loops. The widening operator must ensure that there are no infinite ascending chains. $P \nabla Q$ can be defined as the system of linear inequalities which are satisfied both by P and Q. As the number of inequalities in P and Q is finite and this operator can only reduce or maintain the number of inequalities, termination in a finite number of steps is ensured. More complex definitions of the widening operator may achieve a greater degree of precision [16, 59, 89].

- **Quantifier elimination** ($P[abstract\, x]$): this operation removes all the constraints about a given variable of the polyhedron, while keeping all the implicit constraints about the rest of variables intact. This operation is implemented with the Fourier-Motzkin elimination [67] method, i.e. we update the system of inequalities as follows: First, we add all the possible linear combinations of inequalities with non-zero coefficient in x so the coefficient in x becomes zero. For m inequalities, at most $(m/2)^2$ linear combinations will be added to the system of inequalities. Then, inequalities where dimension d has non-zero coefficient are removed.

Figure 2.6 shows some examples of these operations on convex polyhedra. It should be noted that the convex hull and the widening operator are the only operators that lose precision. All other operators are exact. Non-linear guards and

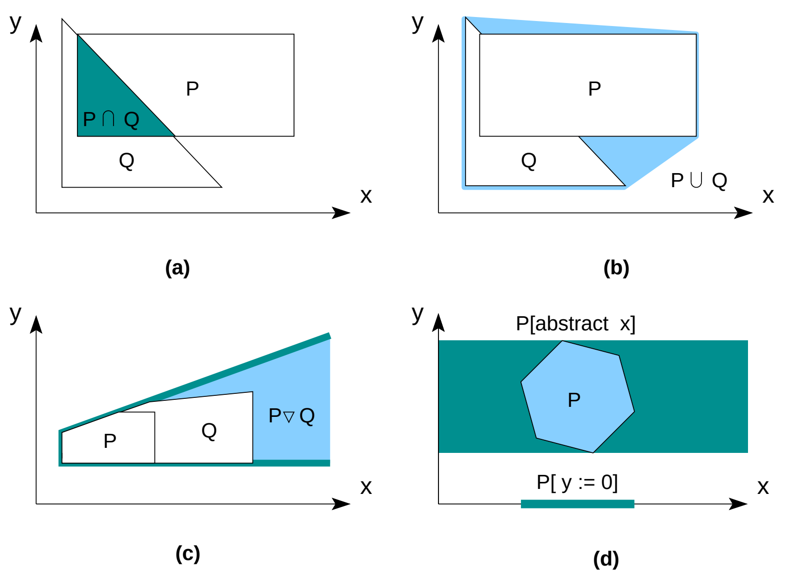


Figure 2.6: Several operations on convex polyhedra: (a) intersection of polyhedra, (b) union of polyhedra as the convex hull, (c) widening of polyhedra and (d) assignment of a linear expression and quantifier elimination.

assignments such as $x := y * z$ are another source of approximation. Typically, non-linear properties are abstracted in some form that can be represented in the convex polyhedra domain such as intervals, e.g. $x := y * y$ becomes $x \leftarrow [0, +\infty]$.

2.4.7 Two-variables per Inequality

The domain of *two-variables per linear inequality* was proposed [144] to improve the precision provided by octagons without the complexity of convex polyhedra. This abstract domain encodes a conjunction of linear inequalities with at most two variables, e.g. $(2x + 3y \leq 3)$. Instead of working with a single polyhedra of n dimensions, this abstract domain works with the $\frac{n \cdot (n-1)}{2}$ projections of the polyhedra onto each pair of dimensions.

The advantage of the two-variable limit with respect to convex polyhedra lies in the convex hull operation. The convex hull of m two-dimensional points can be solved in $O(m \log m)$ time, contrary to the exponential complexity of the same problem in n dimensions, $O(m^{\lfloor \frac{n}{2} \rfloor + 1})$. Moreover, the widening operation sets a constant limit in the number of inequalities that can be stored simultaneously about each pair of dimensions. As a result, there are polynomial time algorithms to perform the operations in this abstract domain. For instance, the closure algorithm that propagates constraints among two-dimension pairs (equivalent to the canonical form computation in DBMs) has a complexity $O(n^3 \log^2 n)$.

2.4.8 Presburger Arithmetic

Presburger arithmetic is the first-order theory of the integers with addition. A Presburger formula may include conjunctions, disjunctions and negations of *linear* inequalities over quantified and free variables, e.g. $\exists x : (3x+y = 7) \vee (x-y \leq 8)$. Nevertheless, it cannot include expressions involving products among variables, e.g $(x^2y = 4)$, or any related operators, e.g. division, modulo and power. Due to this restriction, the validity of Presburger formulas is decidable.

Presburger arithmetic has been used extensively to analyze infinite-state systems outside the abstract interpretation framework, e.g. the Omega library [130, 136]. Its use within the framework has also been proposed with the definition of a widening operator [38, 39].

Contrary to most abstract domains that can manipulate inequalities, Presburger arithmetic can represent non-convex regions without a loss of precision. Therefore, the union operator is exact in this abstract domain. The drawback to this expressive power is the worst-case complexity of the satisfiability algorithm. For a formula of length n, this complexity can be characterized as $O(2^{2^{2^n}})$ [131] or $2^{2^{cn}}$ for some constant c [80]. Even though the worst-case complexity is prohibitive, the complexity in practical examples appears to be tolerable [11, 38].

2.5 Conclusions

Abstract interpretation is a framework for the static analysis of complex systems. In this framework, undecidibility is addressed using abstraction. As a result, the analysis is guaranteed to terminate while the discovered properties are approximate but conservative.

The problems studied in this thesis, the verification of timed systems and infinite-state concurrent systems, are suitable for abstract interpretation analysis. Some reasons why abstract interpretation is appropiate are the following:

- *Theory*: there is a strong theory behind abstract interpretation which ensures the validity of the results obtained with abstract interpretation techniques.
- *Automation*: Analysis based on abstract interpretation are guaranteed to terminate by the underlying theory. Therefore, abstract interpretation analysis can be fully automated.
- *Analysis of numerical constraints*: Abstract interpretation is well-suited for the discovery of properties on numerical variables. In the area of static analysis, there are several known techniques based on abstract interpretation that analyze numerical properties of the variables of a program (see Section 2.4).
- *Trade-off between precision and efficiency*: Abstract interpretation provides a generic framework of analysis. Many different abstract domains can be used to represent the states of a system, each of which provides a different

trade-off between precision and efficiency. For a specific problem, one can select the abstraction with the best trade-off, i.e. the most efficient abstraction with the sufficient precision.

- *Extensibility*: New abstract domains can be "plugged" into the framework of abstract interpretation very easily. Hence, it is possible to build customized abstract domains which are very efficient for a specific problem or a specific class of systems.

The definitions provided in this chapter will be used throughout the rest of the thesis. Precisely, Chapter 4 presents a methodology for the analysis of timed systems with symbolic delays which is based on abstract interpretation. Chapter 5 presents another contribution of the thesis, a new numerical abstract domain called *octahedra*.

Chapter 3

Verification of Timed Systems

It is possible to fail in many ways, while to succeed is possible in only one way.

—Aristotle

This chapter presents related work on the verification of timed systems. The first section is devoted to the modeling and verification of timed systems in general. The second section is devoted to the verification of a special class of timed systems: timed circuits. This previous work is closely related to the contributions presented in Chapter 4.

3.1 Introduction

The correctness of a system can be studied in terms of its inputs, outputs and internal events. In many domains, correctness depends on *what* events occur and the relationship between them, e.g. *"input X should only occur in state Y"*, *"internal event X should not occur"*. However, in real-time applications, correctness also depends on *when* each event happens. Such applications may have constraints on *latency* (*"event X should last no more than 10 seconds"*), *throughput* (*"event X should happen every 2 seconds"*), *separation of events* (*"event X should happen at least 3 seconds after Y"*) or other temporal properties.

3.1.1 Timed and Parametric Timed Systems

A system whose correctness depends on its temporal behavior is called a *timed system*. There are many formalisms that can be used to describe a timed system. In these models, time may appear in different ways: as a delay associated with an event, as a explicit transition that represents time elapsing or as a clock whose value can be read or modified. The properties in these models can be defined and studied using diffent varieties of *temporal logics*. In general, the verification of a temporal property in a timed system has a high computational complexity. In the same way that concurrent systems suffer an *state space explosion* problem, the temporal behavior of a simple timed system can be very complex. To make matters worse, many interesting timed systems are also highly concurrent.

The description of a timed system or a temporal formula may involve values that characterize the temporal behavior. Such values may define the duration of an event, the initial value of a clock or a similar property. To limit the complexity of the analysis, most models define these values as constants, e.g. *"do X until the clock Y reaches 9 time units"*. However, there is a more generic approach which consists on using *parameters* instead of constant values, e.g. *"do X until the clock Y reaches p time units"*. Methods that describe a temporal behavior that depends on parameters are called *parametric* methods. Obviously, the complexity of parametric methods is higher than its timed counterparts. On the bright side, parametric methods can answer very interesting questions which cannot be solved with timed methods. For instance, given a property and a parametric timed system, it is possible not only to check whether the property holds, but also to compute *for which values of the parameters the property holds.*

Section 3.2 is devoted to the description of timed formalisms and temporal logics, together with the complexity results, algorithms and data structures used in their verification. This description includes the methods used in the analysis of parametric timed systems.

3.1.2 Asynchronous and Timed Circuits

Digital circuits can be seen a special case of timed systems. In the real world, logic gates do not produce an output instantly: they need some time to charge/discharge the output signal. The delay of a gate is not constant, as it may vary due to defects in the fabrication, changes in the temperature or the power supply, ... Also, the propagation of a signal in a wire is not instantaneous. Therefore, the timed behavior of a circuit can be very complex and it might have an impact on its correctness.

Timing issues are addressed differently in several design styles. *Synchronous* circuits, for instance, use a global periodic signal called clock to control the behavior of the circuit. Each clock cycle represents the computation of the next state. Even if the computation has been completed before the end of the cycle, next states are not evaluated until the clock allows it. This simplicity makes the behavior very predictable and very easy to verify: if the logic that computes the next state is correct and the clock cycle is longer than its worst-case delay, the circuit will be correct. There cannot be interactions among the computation of different states of the circuit.

On the other hand, *asynchronous* circuits do not use clock signals. Instead, the state of the circuit is only controled by the changes in the input signals and the internal events. The synchronization with the environment is achieved using *handshakes*, which can be implemented using *delay paddings* or additional *completion detection* logic. The verification of asynchronous circuits is complex, as the computation of several states can be interleaved.

Several classes of asynchronous circuits have been defined in the literature. The main difference is the class of delays that the circuit is supposed to accept. Categories like speed-independent (SI), delay-insensitive (DI) circuit or quasi-delay-insensitive (QDI) describe circuits which operate correctly for any delay of the elements, under a specific *delay model*. A different approach is taken by *timed circuits* [122], which are designed to operate correctly only if a set of timing constraints is met. The complexities of the verification of timed circuits are similar to those found in any timed system.

Section 3.3 describes the state of the art in the verification of timed circuits, discussing several types of timing constraints used in the verification process.

3.2 Timed systems

3.2.1 Specification of Timed Systems

There are many formalisms used to specify timed systems. Most of these formalisms are *extensions* of untimed models used to describe finite-state and concurrent systems. The semantics of the extensions varies. In some cases, the formalism establishes that each event has some delay, either defined by a fixed constant or within an interval of constants. Firing an event may depend on these delay constraints, and may imply a passage of time. In other cases, a new class of transition is added to the system, one which does not alter the discrete state, but advances time. The passage of time may be recorded by a set of clock variables whose value can be read by the system. Other temporal semantics are also possible.

The following is a non-exhaustive review of some popular formalisms from the literature. The choice of formalisms is oriented towards the models that are more closely related to the analysis of timed circuits. A more exhaustive survey can be found in [156].

In order to present the different formalisms, we will model several versions of the classic *railroad crossing* problem. In this problem, a train approaches a railroad crossing whose gate is controlled automatically. Several sensors detect the presence of the train as it approaches the gate and also as it leaves the railroad crossing. The controller of the gate should take into account the information from the sensors and open/close the gate at the crossing accordingly. Obviously, the rising and falling of the gate takes some time, and there is some delay between the detection of the proximity of the train and the arrival of the train to the crossing. This information should be taken into account in order to guarantee two properties:

- Safety property: *"whenever the train is at the crossing, the gate is closed"*.
- Liveness property: *"if there is no train in the crossing, the gate will eventually become open"*

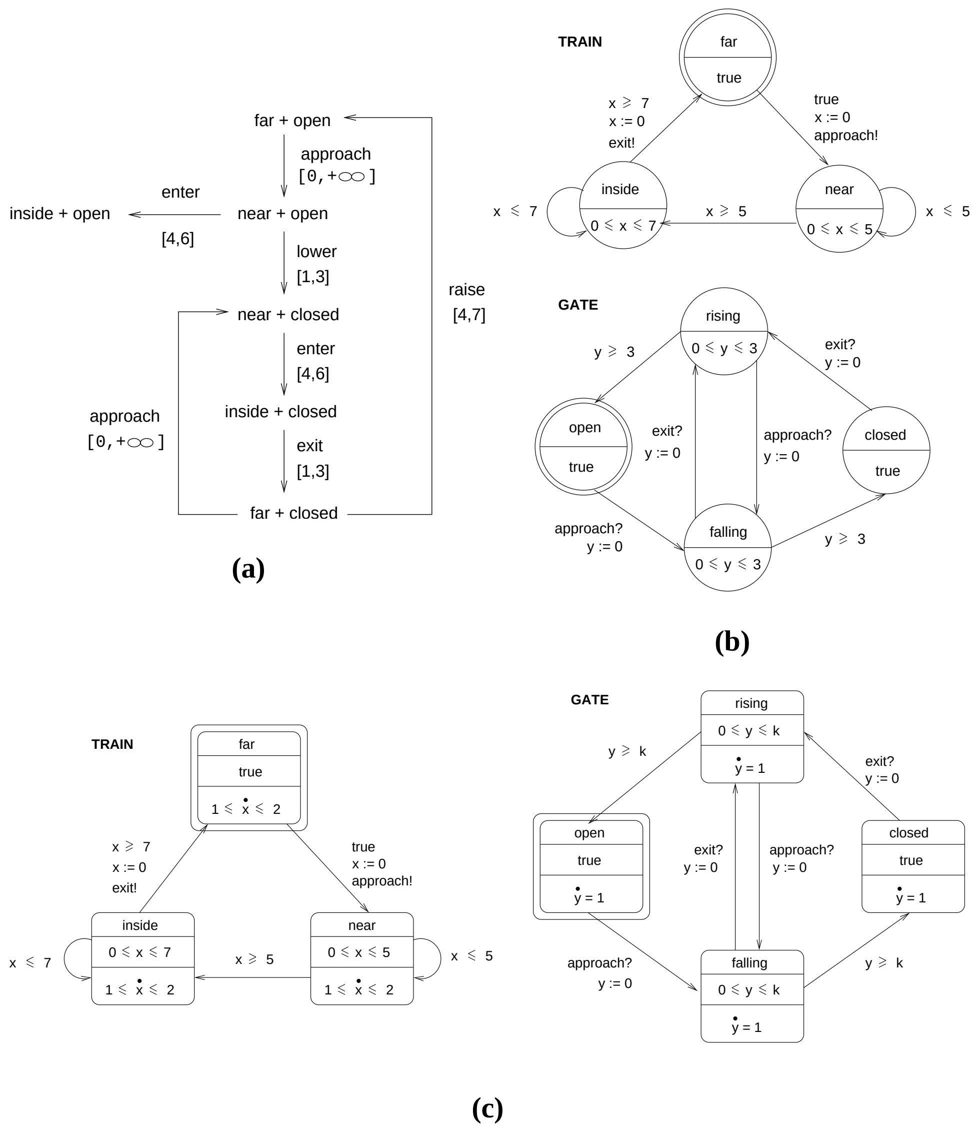


Figure 3.1: The railroad crossing problem modeled in several temporal formalisms: (a) Timed Transition Systems, (b) Timed Automata, (c) Hybrid Automata

Figure 3.1 describes several versions of the railroad crossing problem using different temporal formalisms: Timed Transitions Systems (TTS), Timed Automata (TA) and Linear Hybrid Automata (LHA).

Figure 3.1(a) shows the TTS model. TTS were introduced in [92] as a means to model the timed execution of a set of concurrent processes. Timed transition systems are a extension of the basic computational model of transition systems [14]. In a timed transition system, there is a set of *states*, and in each state, there can be several enabled *events*. In the example, each state describes the position of the train (far/near/inside) and the gate (open/closed). There are three events related to the train (approach the crossing, enter the crossing, and exit the crossing) and two describing the gate commands (raise or lower the gate). Whenever an event is fired, the state is changed according to a *transition relation*, described graphically in the example. The event to be fired is chosen non-deterministically among the enabled events. Each event e has an associated interval of positive real numbers, noted as $[d_e,D_e]$. The lower bound of an event e represents the minimum amount of time that must elapse between the moment that e became last enabled and e was fired. Conversely, the upper bound of e represents the maximum amount of time that can elapse between the last enabling of e and the firing of e. It should be noted that the amount of time is counted since the event became enabled, even if it happened in a previous state. For instance, after firing the event lower, the event enter only takes between 1 and 5 time units to fire, as it has already been enabled between 1 and 3 time units in the previous state.

The TA model is depicted in Figure 3.1(b). Timed automata were introduced in [4] as a formal notation to model the behavior of real-time systems. A TA is an automaton extended with a set of *clock variables* (in the example x and y) and *clock constraints*, comparisons of clocks with constant values. Each state of the automaton, called a *location*, defines an interval that should be satisfied by the clocks while they are in the state. The system may evolve by spending some time in a location or by taking a switch, a transition to another location with a guard to be satisfied and a set of clock variables to be reset. In the example, the railroad crossing is modeled using *communicating TA*: a set of timed automata that synchronize using synchronous communication. A switch may receive a message, noted as *msg?*, and it can also send a message, noted as *msg!*. A switch which sends a message can only be taken if another switch is receiving that message simultaneously.

Finally, a LHA model appears in Figure 3.1(c). Linear hybrid automata are a more powerful formalism for the specification of temporal properties [3, 4]. Like TA, the passage of time is controlled with clock variables. Each location has an invariant which controls the possible values of clocks. Also, each switch has a guard that defines the valid values for clocks and a set of clocks to be reset. But there are important differences with respect to timed automata. First, the guards and invariants may include *parameters*. In the example, the gate requires k time units to be raised or lowered. Also, the set of clock constraints is larger than in TA: any linear inequality over clocks and parameters can be used in the guards and invariants. The last difference is the possibility of describing clocks that run at different *rates*. For example, the timer for the train may move twice as fast as the timer for the gate. This can model, for instance, the fact that the trains going through the gate may travel at different speeds, with some trains traveling twice as fast as the others. Clock rates can be specified on a per clock and per state basis, providing a large degree of flexibility in the specification.

3.2.2 Temporal Logics

A temporal logic is a formalism used to describe properties that involve time. Time may appear in a formula in many different ways. For instance, as a relative order among atoms (*"Y happens after X"*), as a bounded interval (*"X happens between 2 and 4 time units"*) or as a reference to a clock variable (*"X happens while* $b \leq 2$*"*), to cite a few.

In non-temporal logics, a crucial problem is the *satisfiability* of a formula. Meanwhile, a more relevant problem in temporal logics is *model-checking*: *"given a timed system and a temporal formula, check if a model of the system satisfies the formula."*. The "model" and, therefore, the criterion to decide the correctness of formulas in a logic has multiple definitions.

All the previously mentioned choices, combined with several possible combinations of temporal and non-temporal operators, lead to a huge number of temporal logics. Each logic provides different advantages according to expressiveness, decidability and efficiency of the model-checking procedure. The discussion of all these logics is outside of the scope of this thesis, and the interested reader is referred to [6, 76, 156] for a survey. The rest of the section will briefly mention some important classes of temporal logics in order to facilitate the description of the verification techniques.

Linear vs. Branching Time

There are two ways in which a formula can express a property about a timed system. First, the formula might consider the dynamic behavior of the system during a single run, and establish properties which hold in this run. This is known as *linear time* logic, such as Linear-Time Propositional Logics (LPTL) [135]. On the other hand, formulas might attempt to describe the dynamic behavior of the system focusing on the possible runs, e.g. *"for all runs X"* or *"there is a run where X"*. This alternative is called *branching time* logic, and an example is Computational Tree Logic (CTL) [77].

Discrete vs. Dense Time Semantics

Some logics can only describe *qualitative* properties, such as the relative order among atoms (*"X happens before Y"*) or the inevitability of an atom (*"X will eventually happen"*), e.g. LPTL and CTL. On the other hand, other logics can describe

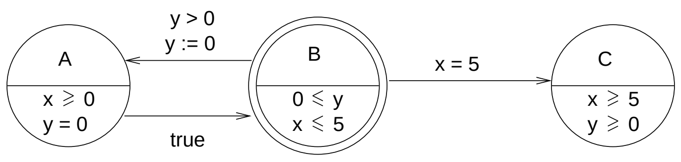


Figure 3.2: A TA with a behavior that depends on the discrete/dense semantics

quantitative properties such as duration (*"X happens for 2 time units"*), like Metric Temporal Logic (MTL) [6]. An advantage of the qualitative temporal logics is that *time is discrete*, i.e. a formula must be checked at some discrete instants of time with no need to study what happens between them.

Quantitative methods may also use a *discrete* semantics, where the smallest possible time elapse is defined and becomes the time unit. The duration of an event or any change in a clock will be described as integer multiples of this time unit. Meanwhile, there is an alternative *dense* semantics where time advances continuously and it is not possible to discretize it. As a result, clock readings and delays may have rational or real values in addition to the integer values from the discrete model.

Example 3.1 *Let us consider the TA in Figure 3.2. Remarkably, the semantics of this TA is different in discrete time and dense time.*

The TA contains three locations A, B and C, with B being the initial location. The location A can only be visited from location B when the clock y is not zero, and clock y is reset afterwards. In the discrete semantics, this can happen only a finite number of times, e.g. is the time unit is 1, location A will be visited at most 5 times. On the other hand, in dense time location A can be reached an infinite number of times. Therefore, the distinction between discrete and dense semantics has an impact on the behavior of the system, and should be chosen carefully.

The relationship between discrete and dense time is a complex issue. The problem is whether there exists a procedure, called *discretization*, which can transform a dense time specification into a semantically equivalent discrete time model. Of course, the existence of a discretization procedure depends on the notion of "equivalence" being used. Early theoretical results discovered that discretization is possible on timed transition systems [93] (equivalence = same qualitative behavior) and timed automata [84] (equivalence = same untimed language). Several restricted classes of hybrid systems can also be discretized [7]. Further work established that discretization of timed automata keeping the same qualitative behavior is only possible for acyclic automata [15]. This result can also be applied to the analysis of asynchronous timed circuits: cyclic (sequential) circuits must be analyzed with a dense-time semantics. More results establishing decidability and complexity of several temporal logics in discrete and dense time semantics can be found in [6].

3.2.3 Analysis of Timed Systems

The following is a brief summary of techniques used in the verification of timed systems. The presentation will focus on the techniques used in the verification, rather than covering the specific temporal logic being used or the properties being verified. Many of these techniques will be mentioned again in the context of timed circuits.

Discrete Time Methods

The analysis of a discrete time model has a big advantage over dense time: between two points in time, the number of instants to be studied is finite. This finite nature of the timed state space is well suited for the use of untimed model-checking techniques [100]. As the timed state space is potentially very large, efficient methods are required to represent it. A common technique is based on *decision diagrams*, the so called *symbolic representation*[1].

Decision diagram techniques have been applied successfully to several problems in different application domains. Binary Decision Diagrams (BDD) [37] provide an efficient mechanism to represent boolean functions. Some strengths of BDDs are the ability to manipulate sets of values symbolically due to its recursive nature, and the canonicity of the representation. In the context of discrete time systems, many verification approaches based on BDDs have been described [25, 34, 75]. However, the size of the state space and thus, performance, depends largely on the magnitude of constant values used in the delays and clock readings. Large values may multiply the size of this state space. These problems can be addressed by using techniques for the dense time semantics in a discrete time model [35].

DBM-based Methods for Dense Time

The study of the dense time model is slightly more complex. According to the dense semantics, there is an infinite number of time instants between two points in time. As a result, the first problem in dense time is the definition of a finite model which encodes these infinite sets of values. The basic element used in this encoding is the *region* or *unit-cube* [2]. A region is a set of clock valuations such that (a) the integer part of all clocks is equal and (b) the order between the fractional parts of the clocks is the same. A compact way to represent regions is using *zones*: a zone is a convex set of regions[2]. Figure 3.3 compares how a set of clock valuations is encoded in the discrete semantics and in the dense semantics as a set of regions or as a single zone.

[1]The term *symbolic* is misleading, as it is used with two senses in the literature. In most references, symbolic denotes the use of a decision diagram data structure, while in other cases it refers to the verification of a parametric system (a system with symbolic delays).

[2]Some references use the term *geometric region* to describe a zone

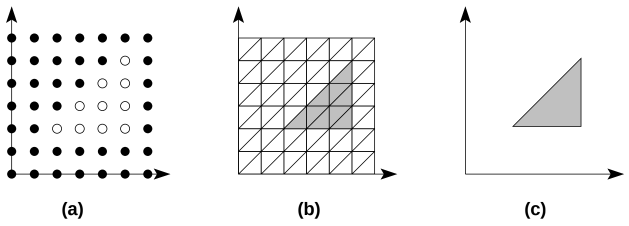


Figure 3.3: Representations of timed states: (a) discrete semantics, (b) dense semantics encoded with regions, and (c) dense semantics encoded with zones

Zones have a very efficient implementation called *Difference Bound Matrix* (DBM) [72]. A DBM is a two-dimensional matrix that encodes a lower and upper bound on each clock and on the difference between each pair of clocks: a zone. The concept of a DBM, together with an example, has been already been presented in Section 2.4.4.

Zone-based methods can use DBMs to compute the timed state space of a dense time specification [23, 33]. However, an important drawback of DBMs is their inability to represent non-convex sets of regions. A direct consequence is the need to store *a set of DBMs* for each untimed state, instead of a single DBM. As a result, the large amount of storage space used by DBMs quickly becomes prohibitive and limits the scalability of these methods. Several strategies can be used to reduce this memory usage. For example, *variable-dimension DBMs* reduce space by representing only non-redundant clocks whose value will be read before the clock is reset [68]. Another approach, the *minimal constraint representation* avoids encoding redundant constraints like $(clk_i - clk_j \leq \infty)$ in the DBM [104]. Also, the information stored in a set of DBMs can be approximated using the *convex hull* like it is done in abstract interpretation [71]. Additional clever implementation decisions [18] can be employed to improve the memory usage even further, but the problem of scalability of DBM analysis still remains. In addition, attempts to represent non-convex sets of regions explicitly exhibit a prohibitive complexity, e.g. timed polyhedra [31] have a space complexity above $O(n!)$.

Decision Diagram Methods for Dense Time

The success of decision diagrams in concurrent and discrete time systems has motivated a lot of research in the application of symbolic methods to the dense time semantics. To perform this verification, there are other decision diagrams than BDDs. For example, Multi-Terminal Decision Diagrams (MTBDD) [82] represent functions from boolean variables to reals, $f : \mathbb{B}^n \rightarrow \mathbb{R}$. This kind of functions can be used to encode numerical matrices, so a MTBDDcan be used to store many DBMs [150], even though the DBMs must be reconstructed in order to operate with them.

Several approaches attempt to perform a fully symbolic timing verification,

Property	**DBM**	**DDD**	**CDD**	**RED**	**CRD**	**NDD**
Dense model of time?	yes	yes	yes	yes	yes	no
Convex set of clock valuations?	yes	yes	yes	yes	yes	yes
Non-convex set of clock valuations?	no	yes	yes	yes	yes	yes
Fully canonical representation?	yes	no	no	yes	yes	yes
Independent of the magnitude of constants?	yes	yes	yes	no	yes	no
Improves memory usage w.r.t. DBM?	–	n/a*	yes	yes	yes	n/a*
Improves CPU time w.r.t. DBM?	–	n/a*	no	no	no	n/a*

* Data not available or very few experiments available.

Table 3.1: Comparison of several data structures for timing analysis

i.e. the zones are stored as some kind of decision diagram where all the required operations can be performed without building a DBM. But there are three problems that must be faced by all these techniques. First, zones can be characterized as constraints on the difference of *two variables*, while decision diagrams are well suited to represent properties *one variable at a time*. Then, a set of difference of pairs of clocks may imply new constraints on other clocks. In DBMs, these implicit constraints are discovered through a closure step which implies a traversal of *the entire* DBM. However, decision diagrams are well suited for operations that can be defined as a *recursive traversal* that operates on one variable at a time. In order to remain efficient, some of these data structures perform only local or restricted closures, while others perform full closure even though it is an expensive operation. Finally, the complexity of DBM algorithms depends only on the *number of clocks*, while in some decision diagram techniques, the complexity will also depend on the *magnitude of the delay constants* that appear in the timed system.

Table 3.1 compares several decision diagram approaches with DBMs. These approaches are the following:

Difference Decision Diagrams (DDD) [115] are a special kind of interpreted BDDs. Instead of being a boolean variable, each node encodes an inequality of the form $(clock_i - clock_j \leq k)$. The *then* and *else* arcs going out of each node describe the state space according to the truth value of the inequality.

Clock Difference Diagrams (CDD) [18,19] encode a difference of clocks of the form $(clock_i - clock_j)$ in each node. There can be several outgoing arcs in a node, each labeled with an interval, with the only condition that all arcs of a node must have disjoint intervals. Each arc is taken if the upper bound of the difference of clocks lies in the specified interval. The terminal nodes, *true* or *false*, specify whether a given valuation belongs or not to the decision diagram.

Clock Restriction Diagrams (CRD) [155] are also similar to CDDs, as they also encode a difference of clocks in each node, and each node can have several outgoing arcs. However, instead of labelling each node with a disjoint interval, each node is labeled only with an upper-bound. In some systems, this

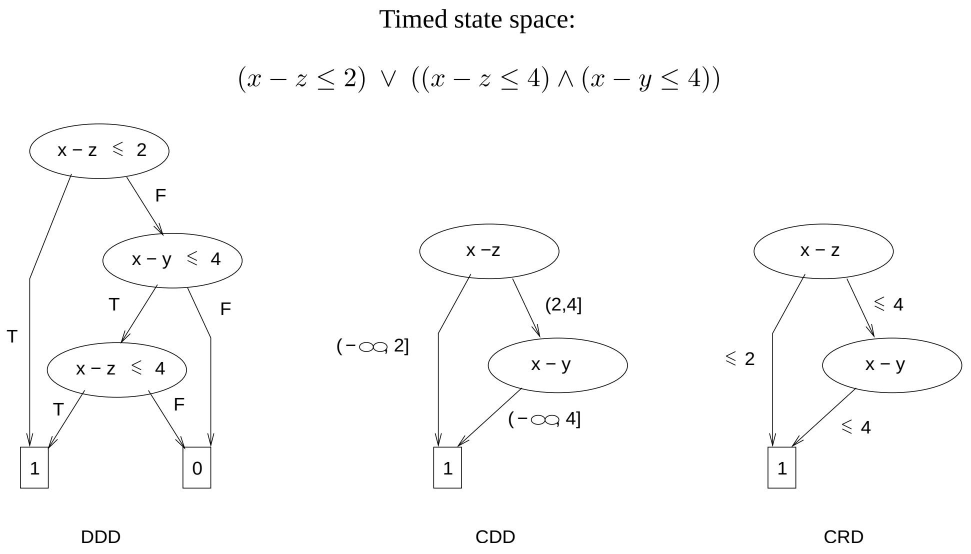


Figure 3.4: A DDD, a CDD and a CRD encoding a timed state space.

improves the performance with respect to CDDs.

Region Encoding Diagrams (RED) [153] encodes one clock in each node. The order of the fractional parts of clock readings is encoded in the ordering of the variables in the decision diagram, while the value of the integer part is encoded in intervals labeled in the edges, like CDDs. The performance of this data structure is very dependent on the value of the constants appearing in the timed system.

Numerical Decision Diagrams (NDD) [3] [75] encode the valuations of clocks as sets of numbers. Each node of the decision diagram considers a bit of a number, and the *then* or *else* branch is chosen depending on the value of the bit. Thus, the efficiency of this approach depends on the magnitude of the constants used in the timed automata, which determines the number of bits used to represent a number.

Even though the experimental results with these approaches provide large savings in terms of memory usage, these savings come at the cost of more CPU time managing the data structures. In examples with very regular/symmetric state spaces, the reductions in memory usage may be huge, and some examples may even exhibit a reduction in CPU time.

Example 3.2 *Figure 3.4 presents several decision diagrams used in the analysis of timed systems. The state space represented by these decision diagrams is not*

[3]NDDs can only be used in the discrete time semantics. They are included in this list to allow a comparison with the other symbolic data structures.

convex, i.e. it cannot be represented by a single DBM. Therefore, using these methods leads to a reduction in memory usage. This reduction is even larger when we consider that several decision diagrams can share information among themselves, i.e. a common subgraph is only stored once.

3.2.4 Parametric Timed Systems

In the timed models discussed so far, the delays of the events and the clock readings are a part of the model. Even when the exact delay is unknown, an interval describing a conservative upper and lower bound must be provided, e.g. in timed transition systems. But a system that satisfies a property with some values of the delays may not satisfy the same property with different delays. This fact points out two important shortcomings of timing analysis. First, it is not possible to check whether a property holds regardless of the value a delay. Also, whenever the delays of the model are modified, timing analysis must be rerun to make sure that the properties hold with the new delays.

Parametric timed systems [5, 8, 78, 154] are an extension of timed systems where some delays, called *parameters* or *symbolic delays*, are not specified in the model. Parametric extensions of the systems described in Section 3.2.1 can be defined trivially. For instance, parametric timed automata may contain parameters instead of constants in the guards and invariants, while parametric timed transition systems may have parametric lower and upper bound for delays. Also, linear hybrid automata allow parameters in the specification. The analysis of these parametric timed system studies how the properties about the model depend on the value of the parameters, e.g. *"property P holds for any value of the parameters"* or *"property P is satisfied if $(p_1 \geq 3) \wedge (p2 < p_4)$"*. Some examples of problems that formalize this analysis are:

Emptiness (decisional): *"Given a parametric timed system and a property, is the property satisfied for some value of the parameters?"*

Parametric verification (decisional): *"Given a parametric timed system, a property and a (possible infinite) set of values of the parameters, is the property satisfied for all values of the parameters?"*

Parametric synthesis: *"Given a parametric timed system and a property, find all values of the parameters for which the system satisfies the property"*.

Regarding these problems, several decidability results have been established for parametric timed automata (PTA). In general, the emptiness problem is undecidable [8], although for some restricted classes of PTAs the problem is decidable. For instance, the number of clocks that are compared to a parameter in some guard or invariant of the PTA is important. The emptiness problem is decidable in PTAs with only one parametrically constrained clock, it is undecidable for automata with three or more parametrically constrained clocks, and decidability is unknown for

PTAs with two such clocks [8]. Another class of PTAs where the emptiness problem is decidable is lower-bound/upper-bound automata (L/U automata) [97], where a L/U automaton is defined as a PTA where all the parameters appear only as lower or upper bounds of the clocks, but not both, in every guard and invariant of the automaton. As a consequence of these decidability results, most analysis of non-trivial parametric timed systems are undecidable.

3.2.5 Analysis of Parametric Timed Systems

Like in timed systems, techniques that verify parametric timed systems emphasize the representation of the clock values, exactly as in timed systems. These clocks can be compared to a parameter, reset to a parameter, or incremented by a parameter or a constant, so the core of parametric timing analysis techniques is an efficient computer representation of constraints among clocks, parameters and constant delays. Contrary to non-parametric timed systems, these constraints may involve more than two variables, e.g. two clocks and three parameters. Therefore, the data structures used to represent zones efficiently in conventional timing analysis cannot be used directly in parametric timing analysis.

Some methods have been proposed to study several decidable problems in parametric timed systems, e.g. model-checking of TCTL formulas with parameters over a timed automaton without parameters [154]. However, most interesting problems are undecidable. In order to face undecidability, two kinds of approaches are used in this area. On one hand, exact approaches whose termination is not guaranteed in general, called *semi-decision* procedures. The advantage of this exact procedures is that they can be used to analyze liveness properties [28]. On the other hand, other approaches use approximate methods whose result may be inexact while being conservative, e.g. like abstract interpretation. These methods are more efficient, but they can only be used to study safety properties. It should be noted that the all these techniques have a *very high computational complexity*. A very important factor that determines this complexity, besides the size of the system, is the *number of parameters* used in the model.

Some related methods which also deal with symbolic delays will be presented in Section 4.6, as they are closely related to the contributions of Chapter 4.

Parametric DBM Methods

Two parametric extensions of DBMs have been proposed [12, 97] to study parametric timed automata. Both approaches are exact, but provide only semi-decision algorithms that might not terminate in general. In these extensions, constraints like $(clk_i - clk_j \leq E)$, where E is an expression over the parameters, can be encoded. Additional constraints over the parameters are attached to the DBM-like matrix, e.g. $(p_1 \leq p_2)$. Each approach supports a different family of expressions.

The technique presented in [12,13] allows non-linear constraints on the parameters, e.g. $(clk_i - clk_j \leq p_1^2 - 2p_2)$. The constraints on the parameters are encoded as a first-order arithmetic formula. Instead of using a widening operator to extrapolate the behavior in loops, additional counter variables are added to the parametric DBMs. These variables keep track of the number of iterations in a loop, so the value of clocks after a loop can be encoded as an expression over the parameters and the counter variables. Although this approach is more exact than widening, it is unclear whether it can be widely applied. Furthermore, the complex polynomial properties arising in this extrapolation can only be solved with mathematical decision procedures that have an extremely high complexity [73].

In contrast, the approach presented in [97] only deals with linear constraints on the parameters. The linear constraints allow the use of a linear programming solver (LPMC) [147] which is more efficient than non-linear solvers. Even then, this method exhibits a very high complexity.

Convex Polyhedra Methods for Parametric Analysis

Linear hybrid automata can be analyzed using convex polyhedra [3, 91]. This method is directly inspired by the linear relation analysis [66] described in the context of abstract interpretation (see Section 2.4.6). As such, undecidability is addressed through approximation.

The contribution presented in Chapter 4 is closely related to these approaches. Convex polyhedra are used to analyze the behavior of a timed system with symbolic delays. The difference is the underlying timed model, a parametric timed transition system instead of a linear hybrid automata.

Decision Diagram Methods for Parametric Analysis

The success of symbolic techniques for the verification of real-time systems (see 3.2.3) has motivated the study of these methods for parametric timed systems. The desired goal is a decision diagram where the constraints on clocks and parameters can be represented and manipulated together with the discrete state.

However, the definition of this data structure is not so simple. First of all, the problems being studied are undecidable in general, so the decision diagram should somehow avoid undecidability. Also, region constraints are not limited to two variables like in non-parametric models. Many of the decision diagrams presented in Section 3.2.3, i.e. DDDs, CRDs and CDDs, took advantage of the two variable limit, so they cannot be used.

The *splitting-trees* used in partition refinement methods [147] are binary trees that encode regions in the leaves, and each level defines a *split*, i.e. a division of the state space according to the truth value of a linear constraint. In some sense, these methods are similar to decision diagrams, but the data structure is a tree instead of a directed acyclic graph (no reduction rules) and the timing information is mostly encoded in the leaves as DBMs. The first proposal of a decision diagram to deal with parametric systems was described in [111]. The data structure was called *Decision Diagrams with Constraints* (DDC), and the aim was the analysis of interpreted automata, a class of parametric counter automata[4]. Each node of a DDC has two children, *true* and *false*, like in BDDs. However, the nodes of a DDC can be either a boolean variable *or* a linear inequality, whose truth value declares whether the inequality holds or not.

Another proposal is *Hybrid Restriction Diagrams* (HRD) [157], a data structure for the analysis of linear hybrid automata. HRDs expand the concept of CRDs for non-parametric timing analysis. Each node is a either a boolean variable with two children or a linear expression (not a linear inequality) with several children; the children of a linear inequality E are labeled with a rational k that defines the inequality $(E \leq k)$.

Both techniques allow a symbolic analysis of parametric systems, i.e. all the analysis is performed on the decision diagram. Also, both approaches can describe non-convex sets of values of parameters and clocks/counters. Undecidability is addressed through approximation: several operations like the emptiness test are not exact, but conservative. However, there are several drawbacks to these techniques.

First, there is no canonical form for these decision diagrams. Furthermore, these techniques do not describe a systematic procedure to combine the linear inequalities constraints that appear in the decision diagram. Instead, heuristic algorithms that combine properties that lie close in the decision diagram are used, e.g. check if a constraints in a node N implies the constraints in the direct descendants and ancestors of N. Another disadvantage is that the set of linear inequalities over a finite set of variables is infinite. As a result, the *depth* of these decision diagrams is not bounded in general, while BDDs, DDDs, CDDs and CRDs all have a bounded depth. Finally, another problem faced by these techniques is the lack of a systematic criterion to establish the ordering among the nodes. There are only some heuristics available to decide which ordering is better [111, 157]. Variable ordering is a crucial factor in the efficiency of decision diagrams [139].

Example 3.3 *The decision diagrams in Figure 3.5 encode the state space of a system defined by the property:*

$$((x - 2y + 4z \leq 2) \ \wedge \ (3x - 4y \leq 3)) \ \vee \ ((x - 2y + 4z > 2) \wedge (3x - 4y \leq 9))$$

Note that DDC and HRD use a different strategy to encode the linear inequalities. On one hand, a DDC benefits from properties where one linear inequality and its complementary appear. On the other hand, HRD can reuse information in properties where several linear inequalities appear with the same coefficients but a different constant term.

An important aspect of this example is the ordering of the nodes. For example, the DDC in the Figure would be larger if the top node was not $(x - 2y + 4z \leq 2)$. Another important aspect these decision diagrams is canonicity. It is not possible

[4]This class of automata is not a timed system, but it is closely related and the techniques used in their analysis are similar.

State space:

$$((x - 2y + 4z \leq 2) \wedge (3x - 4y \leq 3)) \vee ((x - 2y + 4z > 2) \wedge (3x - 4y \leq 9))$$

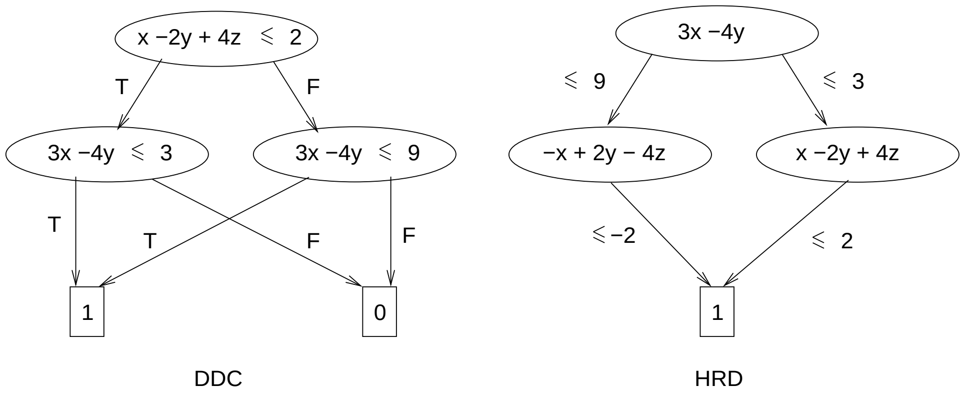


Figure 3.5: A DDC and a HRD encoding a parametric state space defined by a linear assertion.

to obtain a canonical representation in the decision diagrams. For instance, the property that describes the state space can be simplified as:

$$((x - 2y + 4z \leq 2) \ \wedge \ (3x - 4y \leq 3)) \ \vee \ ((3x - 4y \leq 9))$$

This means, for instance, that the HRD in the example could be simplified. There is no systematic procedure to perform these reductions, as well as there is no systematic procedure to choose the optimal ordering. However several heuristics can help to obtain good orders and remove unnecessary nodes from the decision diagrams.

3.3 Timed Circuits

3.3.1 Introduction

In the physical implementation of a circuit, the propagation of signals through wires and the computation performed by logics gate is not instantaneous. Each element has an associated delay that should be considered during the design process. These timing issues are addressed differently in each design style. In synchronous design, a clock signal divides the computation in *cycles*, such that the results from the current cycle are not propagated until the end of the cycle. Figure 3.6 shows an overview of this design style. A circuit can be seen as a *state machine*. At a given point in time, the output of the circuit depends on the inputs and, possibly, on the previous sequence of inputs which defines the current *state*. Thus, the logic of the circuit performs two tasks: computing the outputs and the next state. The output and state signals are stable until the end of the cycle, when the new values are loaded into the registers. There is no early completion: even if the result is

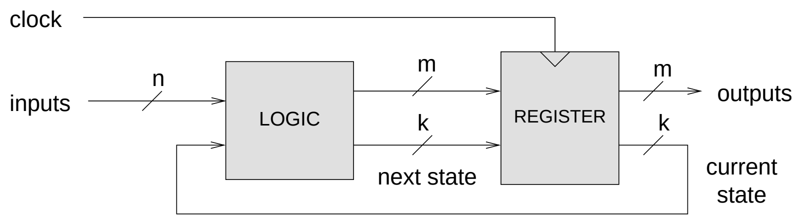


Figure 3.6: Overview of a synchronous circuit.

available before the end of the cycle, it does not propagate forward. This separation avoids potential problems in the feedback loops within a circuit. In terms of correctness, the only challenge is setting the period of the clock in such a way that the length of each cycle is longer than the worst-case delay of the circuit.

In asynchronous design, the lack of a global clock signal for synchronization makes the correctness of the circuits more dependent on the delays of internal and external events. Several classes of asynchronous circuits attempt to deal with delays in the more generic way possible: designing circuits which behave correctly regardless of the delay. Speed-independent (SI), delay-insensitive (DI) circuit and quasi-delay-insensitive (QDI) approaches use different assumptions to generate a circuit which works correctly for any delay. However, this genericity can restrict the set of synthesizable functions, and leads to implementations with more area and delay.

A more aggressive design style is that of *timed circuits* [122], which embrace the dependency between correctness and delay. The goal of timed circuits is obtaining a design which operates correctly under a set of delay assumptions called *timing constraints*. These timing assumptions are used to simplify and optimize the logic that implements the circuit. In this way, an implementation with a lower area and delay can be achieved [58, 123], although its behavior is undefined when the timing constraints are not satisfied. Counteracting the improved performance, verification is required to check that the circuit operates correctly under the timing constraints.

Example 3.4 *Let us consider the circuit presented in Figure 3.7, where the shaded area highlights a feedback loop. In synchronous terms, this feedback loop is computing the* next state *information. Due to the lack of clock signals, this loop may cause undesired behaviors. For instance, the feedback loop may be too fast, overwriting the current state with the next state before the outputs can be produced. Conversely, the feedback loop may be too slow, so that the circuit produces an output that leads to a change in the inputs change before the computation of the next state is completed. It is necessary to impose timing constraints that ensure that these scenarios cannot occur.*

The problem of verification of a timed circuit can be formulated in two versions: a *decisional* version and a *synthesis* version. The decisional version is

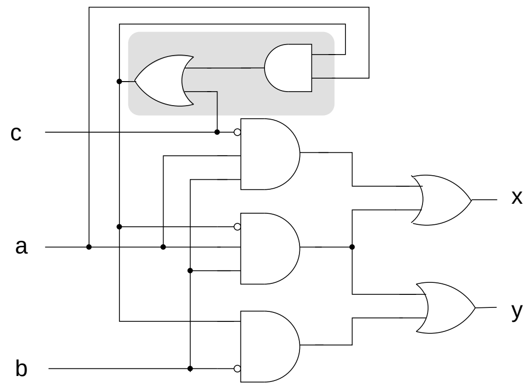


Figure 3.7: An example of a timed circuit.

"given a timed circuit, a specification, and a set of timing constraints, decide if the circuit satisfies the specification when the timing constraints are satisfied". This version is useful when timing constraints are discovered during the design process or when timing constraints are studied manually. Meanwhile, the synthesis version of the problem can be stated as *"given a timed circuit and a specification, discover the timing constraints required to satisfy the specification"*. Finally, a problem related to verification is the *validation* of timing constraints: *"check, after placement and routing, that the real delays of the circuit fulfill the timing constraints"*.

There are several strategies to employ verification during the design of a timed circuit. The main difference between the methods is: (a) whether the timing constraints are available during logic synthesis and (b) the type of timing constraints that are supported.

There is a chicken-egg problem between logic synthesis and verification. On one hand, the timing information is generally not available until logic synthesis is performed. However, at that point, verification can only guarantee a correct operation by modifying the circuit adding extra delay paddings. On the other hand, logic synthesis could use the information from the timing constraints to optimize the circuit [58,123]. Nevertheless, timing constraints can only be available initially if they are somehow described on the specification, can be derived using conservative estimation or can be manually discovered by the designers. This process might be too conservative or it might be difficult to automate.

In the literature, several families of timing constraints are described. Each family provides different advantages: precision of the constraints, available methods to solve the decisional/synthesis verification problem, or simple validation of the timing constraints.

Metric timing constraints [21, 42, 106] specify a lower and upper bound for the duration of an event or the time between two events, e.g. *"the delay of event A is within $[2, 7]$ time units"*.

Relative timing (RT) constraints [58,102,133,148] enforce a relative order among

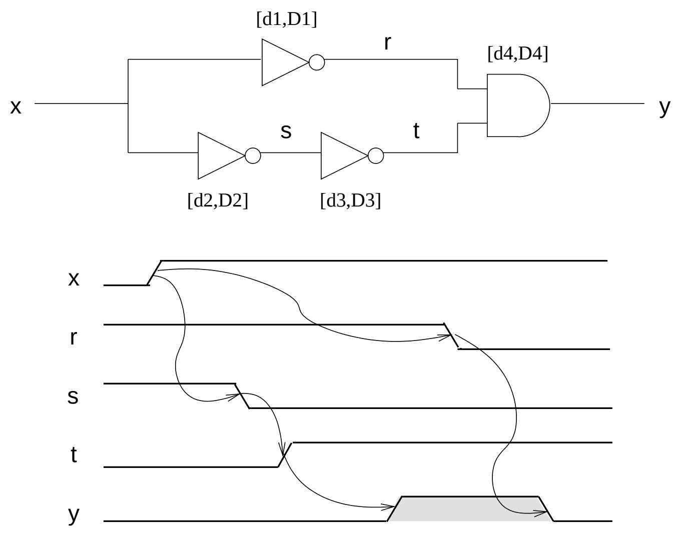


Figure 3.8: Trivial timed circuit used to illustrate different families of timing constraints. On the bottom, a timing diagram highlighting an incorrect behavior.

concurrent events, e.g. *"event A should occur before event B"*.

Chain constraints [125] establish the relative delay between two sequences of events that are triggered by an initial event, e.g. *"after event A, the ordered sequence of events BC should be faster than the ordered sequence DEF"*.

Example 3.5 *Let us consider the trivial circuit depicted in Figure 3.8. We will use it to illustrate the different types of timing constraints. Let us assume that the property to be proved is that* "the output y is always set to false". *To simplify the example even further, let us consider that wires have negligible delay and that the input signal (x) will not change before the circuit becomes stable, i.e. all gates have a correct output according to its inputs. The delay of each gate is unknown, but lower (d) and upper (D) bounds can be established.*

Intuitively, the circuit behaves correctly as long as the two branches have a comparable speed. If one branch is significantly slower than the other, then the output might temporarily fluctuate to one. This scenario is depicted in the timing diagram in Figure 3.8: if the pair of inverters and the AND gate are faster than the other inverter, the output might be set temporarily to one. The following sections will discuss different examples of timing constraints that forbid this behavior.

The approach presented in this thesis models the delays using parameters and discovers the timing constraints on these parameters that are required for correctness. These constraints are defined as linear inequalities that should be satisfied by the parameters. Abstract interpretation is the underlying method used in the verification algorithm. Stated briefly, this method addresses the synthesis verification problem of sequential timed systems with bounded symbolic delays. This methodology has the following *advantages* with respect to the previously presented methods:

- The choice of delay bounds required in metric timing methods is typically made *conservatively*. This is required to guarantee that the delays of the design can be satisfied by the implementation of the circuit. Instead, our approach *avoids conservative choices of gate delays* before the verification by putting off the selection of gate delays until the timing constraints are known. Moreover, if some delay information is known, it can be used to accelerate the analysis.

- Metric timing methods provide results which are only valid for the particular delays chosen in the circuit. Instead, the results of our method are *valid for any delay of the gates*. This effectively means that each circuit must only be verified *once* compared to *once per each delay assignment*.

- Linear inequality timing constraints have several advantages with respect to the timing constraints computed by other methods. For example, linear inequalities are *more precise than metric timing constraints*, leading to more freedom in the choice of delays. Unlike relative timing constraints, it is *very easy to check if a circuit satisfies them*: just check if the delays of the implementation satisfy the inequalities.

- Linear inequality timing constraints provide the guidelines to choose delays for the gates, but allow some degree of freedom. As long as the constraints are satisfied, any delay of the gates will guarantee a correct operation. Therefore, designers can use these timing constraints in order to *select the gate delays* that achieve a good performance, while still making sure that the circuit will be correct.

- Previous methods using symbolic delays had several limitations. Either the circuits were limited to combinational circuits (no feedback loops allowed) or only the decisional verification problem was addressed (the synthesis problem was not automatic).

- Finally, linear inequality constraints have a very *intuitive meaning*: a path in circuit that should be faster than another path. This information can provide useful feedback to the designers of the circuit.

The main *shortcoming* of the method proposed in this thesis is its high computational complexity, which is higher than metric timing constraint methods. As a result, the method is applicable to small or moderately sized timed circuits.

3.3.2 Metric Timing

Metric timing constraints characterize delays with an interval of constant values denoting a lower and upper bound, e.g. "the delay of event A is within $[2,3]$ time units". In order to use metric timing constraints, a description of the timed behavior of the circuit with bounded delays is required. This description requires knowledge of the delays of the internal gates, wires (if the wire delay model is used) and external events. As mentioned previously, these delays may be known as conservative estimates or may be available after the logic synthesis phase. Several techniques work directly on the network of gates annotated with delay information. However, other gate-level techniques model the circuit using a different timed formalism. Some examples of such formalisms are *timed Petri nets* [94, 121], *timed automata* [108], *timed event/level structures* [21] and *process graphs* [96].

The verification of a timed circuit with metric timing constraints is studied by different techniques. First, *time separation of events* computes a lower and upper bound for the time elapsing between two events [41, 96]. Another technique, *min-max timing simulation*, determines a lower and upper bound for the signal propagation delays [40, 70]. The verification problem in its most general version is called *timing verification* or *timing analysis*: given a timed circuit and a set of metric timing constraints, check if the circuit is correct for these constraints.

Timing verification can be addressed by computing the timed state space of the timed circuit. Each event with a bounded delay has an associated clock that keeps track of the amount of time that the event has been enabled. The sets of values of these clocks are stored as geometric *zones*, like it is done for timed automata. The problem lies in the complexity of the approach which can be doubly exponential: once due to the state explosion problem from the concurrency in the circuit, and again due to the complexity of representing time. Several approaches can be used to alleviate this complexity: partial order reductions [20, 159], representations based on decision diagrams [95] or abstractions of the internal nodes of the circuit [127, 160, 161].

Regarding the relationship between verification and synthesis, some techniques can guarantee hazard-freedom after synthesis by adding delay paddings to some signals in the circuit [105]. Other approaches can use metric timing information, obtained from the specification or conservative estimations, to guide the synthesis and optimize the timed circuit [120].

The main drawbacks of these techniques are computational complexity and the need to specify lower and upper delay bounds, which may be overly conservative in some cases.

Example 3.6 *Let us consider the timed circuit in Figure 3.8. In order to use metric timing methods, we need to define constant lower and upper bounds for the delays of the gates. Then, the correctness of the circuit can be checked for these specific delays, like for instance:*

$[d1, D1]$	$[d2, D2]$	$[d3, D3]$	$[d4, D4]$	*Correct?*
$[1, 3]$	$[1, 3]$	$[1, 3]$	$[2, 5]$	*No*
$[1, 3]$	$[1, 3]$	$[1, 3]$	$[7, 8]$	*Yes*
$[4, 6]$	$[1, 1]$	$[1, 2]$	$[1, 4]$	*No*

In this case, the second delay assignment ensures that the output is the constant zero, as the AND gate is slow enough to guarantee that the inverters stabilize before changing the output. The other two delay assignments may temporarily set the output to one.

Verifying the circuit with a set of delays does not reveal whether the circuit is correct with different delays. To check that, a new verification should be performed starting from scratch.

3.3.3 Relative Timing (RT)

Relative timing (RT) constraints enforce a relative order among two concurrent events in a timed circuit. A RT constraint is of the form $A \prec B$, which means that "A should occur before B". Contrary to metric timing methods, RT constraints can be used with an unbounded delay model so several drawbacks of metric timing constraints (need of conservative estimations, complexity) are addressed.

Relative timing was introduced in the design of the RAPPID asynchronous instruction decoder [138]. The generation of RT timing constraints, initially done manually, was automated into a metric timing analysis tool [148]. Other approaches have also been able to compute a sufficient set of RT constraints that guarantees the correct operation of a timed circuit [102,133]. Finally, a methodology that uses RT constraints during logic synthesis to optimize the circuit has been described [58].

Nevertheless, RT cannot replace metric timing. First, there are timing constraints that cannot be represented as RT constraints, such as any metric timing constraint. Also, at some point after placement and routing the RT constraints should be validated. This validation can be performed using simulations, but a complete validation requires using metric timing techniques. Therefore, the validation of RT constraints does not scale to the same extent as the verification.

Example 3.7 *Let us revisit the timed circuit in Figure 3.8. In order to ensure that the output of the circuit in Figure 3.8 is always zero, the inputs of the AND gate should stabilize before the AND gate changes its output to zero. If we denote with $s+$ the transition of a signal s from 0 to 1 and with $s-$ the transition from 1 to 0, the following timing constraints are sufficient to ensure the correctness:*

$$r- \prec y+ \qquad \text{"signal } r \text{ should fall before } y \text{ rises"}$$

$$t- \prec y+ \qquad \text{"signal } t \text{ should fall before } y \text{ rises"}$$

This verification can be performed without having to assign lower and upper bounds to the delays of the gates. When specific delays are given the gates, we will have to validate that these delays satisfy the timing constraints, which requires using metric timing methods.

3.3.4 Chain Constraints

Chain timing constraints compare the delay of two competing sequences of events. Chain constraints have the form "after A, the ordered sequence $B_1 \dots B_n$ is faster than the ordered sequence $C_1 \dots C_n$". In some sense, chain constraints are closely related to RT constraints because they are expressing "after A, $B_n \prec C_n$". However, note that a chain constraint only restricts traces where the sequences occur in the specified order and no event outside the sequences occurs.

Like the RT approach, chain constraint analysis does not require metric timing information. Instead, timed circuits and chain timing constraints are represented using a formalism called *process spaces* [125]. Available methods are able to check whether a set of chain constraints guarantee the correct operation of a circuit [124–126]. The generation of chain constraints is not fully automated, but counterexample traces useful to select new chain constraints can be provided.

Regarding performance, chain constraint methods are only compared to tools applicable to speed-independent designs. Compared to metric timing methods, it appears that chain constraints have a big advantage as they can represent states symbolically (BDDs) and they don't have to represent the timing regions necessary in metric timing. Nevertheless, metric timing constraints cannot be represented as a chain constraint. Regarding RT methods, there is no available data comparing the performance of RT and chain constraint methods.

Compared to the method presented in this thesis, chain constraint analysis is more efficient but has several drawbacks. First, this method only addresses the decisional verification problem (the synthesis problem is not solved automatically). Also, redundant constraints are not detected and simplified automatically. Finally, it cannot take advantage of metric timing information in the analysis, even if this information is available in the specification.

Example 3.8 *The following chain constraints are sufficient to ensure that the output of the circuit in Figure 3.8 is the constant zero:*

$$\begin{array}{rcl} \textit{After } x-, \quad \mathrm{delay}(r+, y+) & > & \mathrm{delay}(s+, t-) \\ \textit{After } x+, \; \mathrm{delay}(s-, t+, y+) & > & \mathrm{delay}(r-) \end{array}$$

These constraints ensure that the output of the circuit is the constant zero.

3.3.5 Timed Circuits with Symbolic Delays

There are several methods which have incorporated symbolic delays in the verification of a timed circuit. These methods will be described in detail in the Section 4.6 because of their direct connection to the contributions presented in the following chapter.

As a short summary, the techniques used in this domain are similar to the ones used to verify parametric timed systems (see Section 3.2.5), such as linear programming or Presburger arithmetics. Again, the complexity of the methods is heavily dependent on the number of symbolic delays that appear in the model. Some methods attempt to reduce this complexity by assigning constant values for some parameters, or studying acyclic circuits. Even then, the efficiency of these methods is below metric and relative timing, and the size of timed circuits that can be analyzed with them is significantly smaller.

Example 3.9 *Let us consider again the timed circuit in Figure 3.8. If the consider the delays d1, . . . , d4 and D1, . . . , D4 as parameters of the problem, the correctness of the circuit can be established if the following constraints on the parameters hold:*

$$\begin{aligned} d1 + d4 &> D2 + D3 \\ d2 + d3 + d4 &> D1 \end{aligned}$$

These constraints provide a generic description of the requirements for a correct operation.

3.4 Conclusions

This chapter has presented the modeling and verification of timed systems, with a special emphasis on asynchronous timed circuits.

Timed systems can be modeled with many different formalisms, like timed Petri Nets, timed transition systems, timed automata or hybrid automata. Temporal properties can be represented using a temporal logics, and model-checking algorithms can test whether a property is satisfied by a timed system. The verification of timed systems is more complex than in untimed concurrent systems due to the complexities of dealing with time. There are may different semantics describing time, e.g. dense/discrete, and therefore many different approaches for verification. Some techniques use Difference Bound Matrices (DBMs) or decision diagrams inspired by DBMs to explore the timed state space efficiently.

A special class of timed systems, parametric timed systems, allow the existence of undefined symbols in the specification. Verification in this context can test the values of the parameters which make the timed system satisfy the property. However, this kind of verification problems is undecidable in general. Semi-decision procedures and approximation techniques can be used to cope with undecidability. Several methods based on convex polyhedra, linear programming and parametric DBMs have been proposed, but all with a high computational complexity.

Regarding the verification of timed circuits, several approaches have been proposed to deal with the complexity of verification, among others, metric timing and relative timing. However, timing constraints which are meaningful, more useful and easy to validate can be derived if the circuit is studied with parametric delays. The next chapter will present one of the contributions of this thesis: a method for the analysis of timed circuits with symbolic delays.

Chapter 4

Verification with Symbolic Delays

If time be of all things the most precious, wasting time must be the greatest prodigality.

—Benjamin Franklin

This chapter describes a new methodology for the verification of timed circuits which uses a more general class of timing constraints. In addition to a bounded interval, the delay of an event can be modeled with an interval of *parameters* called symbolic delays. We present a verification algorithm capable of automatically generating a set of timing constraints on the symbolic delays sufficient to guarantee correctness. These timing constraints are defined as a system of linear inequalities on the symbolic delays.

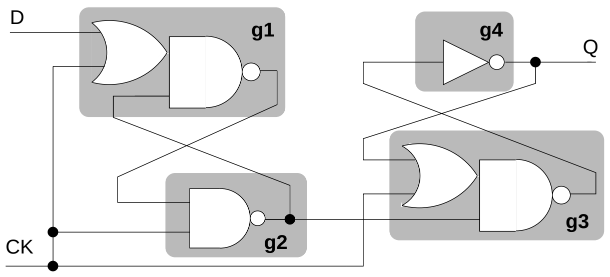
The work presented in this chapter is based on the results published in [48, 49, 52].

4.1 Introduction

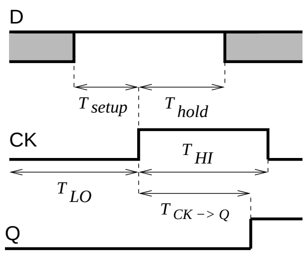
Timed circuits have been introduced in Section 3.3 as as a family of circuits whose correctness depends on timing constraints. Several approaches, such as metric timing or relative timing, can be used to verify the correct operation of a timed circuit. An analysis can also be made in a parametric way, without making any assumption on the delays of the elements.

This chapter describes an algorithm for the verification of gate-level timed circuits using symbolic delays. The goal of this analysis is the *fully automatic* discovery of timing constraints, represented as linear constraints on the symbolic delays, that guarantee the correct operation of the circuit. The timing analysis algorithm is based on the linear relation analysis performed in abstract interpretation (see Section 2.4.6).

The remaining of this chapter will focus on the formalization of the problem, the description of the analysis algorithm and the experimental results obtained on



(a)



(b)

$$\begin{aligned}
T_{CK\to Q} &\leq D_2 + D_3 + D_4\\
T_{setup} &> D_1 + D_2 - d_2\\
T_{hold} &> D_2 + D_3\\
T_{HI} &> D_2 + D_3 + D_4\\
T_{HI} &> T_{hold}\\
T_{LO} &> T_{setup}\\
d_1 &> D_2
\end{aligned}$$

(c)

$$\begin{aligned}
T_{CK\to Q} &\leq 3 + D_3 + D_4\\
T_{setup} &> 9\\
T_{hold} &> 0\\
T_{HI} &> 3 + D_3 + D_4\\
T_{LO} &> T_{setup}
\end{aligned}$$

(d)

Figure 4.1: (a) Implementation of a D flip-flop [134], (b) description of variables that characterize any D flip-flop and (c) characterization of the flip-flop for any delay of the gates, (d) characterization of the flip-flop if some delays are known: $g1 = [4, 7]$ and $g2 = [1, 3]$.

real timed circuits.

Example 4.1 *We illustrate the power of the verification with symbolic delays by means of an example. Let us take the D flip-flop depicted in Fig. 4.1(a) [134]. Each gate g_i has a symbolic delay in the interval $[d_i, D_i]$. We call T_{setup}, T_{hold} and $T_{CK\to Q}$ the setup, hold and clock-to-output times, respectively. T_{LO} and T_{HI} define the behavior of the clock. Our goal is to symbolically characterize the latch behavior in terms of the internal gate delays.*

The method presented in this chapter is capable of deriving a set of sufficient linear constraints that guarantee the correctness of the latch's behavior. The verified property is the following:

> The value of Q after a delay $T_{CK\to Q}$ from CK's rising edge must be equal to the value of D at CK's rising edge.

Any behavior not fulfilling this property is considered to be a failure. Fig. 4.1(c) reports the set of sufficient timing constraints derived by the algorithm. The most interesting aspect of this characterization is that it is technology independent.

As an example, let us focus on two constraints. First, $d_1 > D_2$ is necessary to prevent the cross-coupled gates $g1$ and $g2$ read the wrong value of D or enter *metastability. Second, $T_{setup} > D_1 + D_2 - d_2$ defines the setup time that, interestingly, depends on the variability of the delay of g2. In case of no variability on the delays, the constraint is reduced to $T_{setup} > D_1$, which is the time required for $g1$ to capture the value of D.*

The degree of parametrization can be chosen at the designer's will. If some delays are known, they can be used during the verification. As an example, let us assume that the delay of g_1 and g_2 are in the intervals $[4, 7]$ and $[1, 3]$, respectively. The sufficient constraints with these assumptions are reported in Fig. 4.1(d).

4.2 Formalization of the problem

4.2.1 Basic Notation

The implementation of a timed circuit can be described as network of *gates G* connected by a set of *signals W*. Signals will also be called *wires*.

Regarding the gates of the circuit, we will study the circuit at the *gate-level*: we assume that the circuit is decomposed into simple logical gates. However, the technique can also be applied to a *complex-gate* level analysis.

Regarding the signals of the circuit, each signal x can have two truth values: true, noted as x, and false, noted as $\overline{x}$. Some signals will be labeled as *input* and *output* signals. In some technologies, a signal can be used both as an input and as an output, e.g. [149], but frequently input and output signals are distinct.

A *state* of the circuit is an assignment of truth values to all the signals of the circuit, i.e. a mapping $c : 2^S \rightarrow \mathbb{B}$. The term *event* will be used to denote a change in the value of a signal. Depending on the location of the signal, events will receive a different name. Changes in the value of an input signal will be called *input events* or *environment events*. If the signal is an output of the circuit, the event will be called an *output event*. Finally, changes of internal signals will be called *internal events*.

Events can also be classified according to the value of the signal. A transition from 0 to 1 is called a *rising* event, while a transition from 1 to 0 is a *falling* event. Given a signal x, a rising event will be denoted[1] as $x+$, while a falling event will be denoted as $x-$.

Each logic gate has one or more inputs and a single output whose value is defined a boolean function of the inputs. The operation of a gate takes some delay due to technological constraints: the truth values 0 and 1 are encoded as different voltages in a wire, and changing the voltage in a wire requires some time while it charges or decharges. For that reason, in a given state it is possible that the output of a gate does not match the expected value of the function for the current inputs. In this scenario, the gate is said to be *enabled* in the current state. We say that the gate is *fired* when the output finally changes to match the expected result. If a gate does not need to modify its output in a given state, we say that it is *disabled* in

[1] An alternative notation used in the literature is $x \uparrow$ for rising events and $x \downarrow$ for falling events.

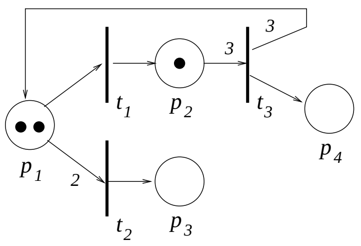


Figure 4.2: Example of a Petri Net.

that state. Firing a gate implies a change in the value of the wire, i.e. a rising or falling event in the output signal. The terms *enabled*, *disabled* and *firing* will also be applied to these events in connection with a gate, e.g. if a gate is disabled, the rising and falling events of its output signal will be disabled.

4.2.2 Interaction with the Environment

The timed circuit does not operate in isolation: an *environment* is interacting with the circuit through the input and output signals. We assume that a specification of the expected interaction between the circuit and the environment is available. This specification should describe the initial value of the inputs and outputs, the valid changes in the inputs, and how the outputs should be changed in response to the inputs.

A possible formalism to describe this specification is that of *Signal Transition Graphs* (STG) [47], a Petri Net model [119]. In order to describe this formalism, some brief notions on Petri Net theory will be introduced.

Definition 4.1 (Petri Net) *A* Petri Net *is a 4-tuple* $PN = \langle P, T, F, M_0 \rangle$ *where* $P = p_1, \ldots, p_n$ *is a finite set of* places, $T = t_1, \ldots, t_m$ *is a finite set of* transitions, $F : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ *is the* flow relation *and* $M_0 : P \rightarrow \mathbb{N}$ *is the* initial marking. *A* marking M *is defined as another function* $M : P \rightarrow \mathbb{N}$*, where* $M[p]$ *denotes the number of* tokens *at* p *in* M*.* M_0 *is the initial marking.*

A Petri Net can be represented by a directed bipartite graph, where an edge $[u, v]$ exists if $F(u, v)$ is positive, which is called the weight of the edge. Graphically, transitions are typically represented by bars or boxes, while places are represented by circles. The tokens inside a place in a given marking are drawn as dots inside the circle for that place.

Example 4.2 *Figure 4.2 depicts the graphical notation of Petri Nets. This example has four places,* p_1*,* p_2*,* p_3 *and* p_4*, and three transitions,* t_1*,* t_2 *and* t_3*. The initial marking is* $p_1 = 2, p_2 = 1$ *and* $p_3 = p_4 = 0$*. In this marking, transitions* t_1 *and* t_2 *are enabled, while transition* t_3 *is disabled. If transition* t_1 *fires, the new marking will be* $p_1 = 1, p_2 = 2$ *and* $p_3 = p_4 = 0$*.*

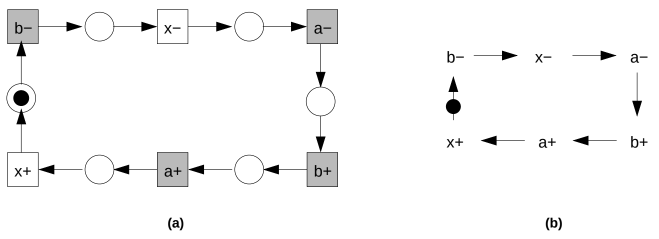


Figure 4.3: Example of a STG using (a) the normal Petri Net notation or (b) the compressed notation for STG.

A transition t is said to be *enabled* at a marking M if $M[p] \geq F(p,t)$ for all $p \in P$. In this case, one may fire the transition at the marking, which yields a marking M' given by $M'[p] = M[p] - F(p,t) + F(p,t)$ for each $p \in P$. A marking M' is said to be *reachable* from M if there is a sequence of transitions fireable from M that leads to M'. The state of a Petri net evolves from the initial marking by firing enabled transitions.

Definition 4.2 (Signal Transition Graph (STG) [47]) *A* Signal Transition Graph *is a triple $\langle N, \Sigma, \Lambda \rangle$ where $N = \langle P, T, F, M_0 \rangle$ is a Petri Net, S is the set of signals and $\Lambda : T \rightarrow S \times \{+, -\}$ is the labeling function that assigns one rising or falling event to each transition.*

This formalism captures the expected interaction between the circuit and the environment. A marking of the STG assigns a value to the signals, i.e. the initial marking defines the initial value of the signals. If a transition labeled with an input event I is enabled in a marking M, then it means that the environment *may produce* event I. On the other hand, if a transition labeled with an output event O is enabled in a marking M, then the environment *expects* the output event O to be produced by the circuit.

The graphical convention used to depict an STG is slightly different from that of regular Petri Nets. Transitions are not depicted as bars of boxes, but as the event with which they are labeled. Places with a single successor and predecessor not displayed. Instead, they are compacted as a single edge over which the tokens may be drawn.

Example 4.3 *Figure 4.3 shows an example of a STG using the typical graphical notation for Petri Nets (a) and the more compressed version used in graphical notation. In this STG, there are three signals, a, b and x. Let us consider a and b as input signals and x as an output signal.*

The initial marking establishes the initial value for the signals: abx. In this initial state, the environment may only change the value of the input b. Then, it expects the output signal x to fall. After that, it can modify the input signals a and b, and again a, in that order. Finally, the environment expects signal x to rise in order to return to the initial state.

Although it is not illustrated by this example, a STG can also describe more complex interactions such as concurrent behavior, e.g. events $a+$ and $b+$ in any relative order, and non-deterministic choices, e.g. either event $a+$ or event $b+$.

4.2.3 Composing the Implementation and the Specification

The behavior of the circuit can be described as a *composition* of (a) the implementation of the circuit as a network of gates and (b) the description of the interaction with the environment. This composition defines a *transition system* that relates the states and events of the circuit.

Definition 4.3 (Transition system [14]) *A* transition system *(TS) is a quadruple $A = \langle S, \Sigma, T, s_{in} \rangle$, where S is a non-empty set of* states, *Σ is a non-empty alphabet of* events, *$T \subseteq S \times \Sigma \times S$ is a* transition relation, *and s_{in} is the* initial state. *Transitions are denoted by $s \xrightarrow{e} s'$. An event e is enabled at state s if $\exists s \xrightarrow{e} s' \in T$. We will denote the set of events enabled at state s by $\mathcal{E}(s)$.*

Intuitively, in this underlying TS a state describes both the values of the signals of the circuit and the marking of the STG. An internal or output event is enabled in a given state if the corresponding gate is enabled in the circuit. Meanwhile, input events are enabled in a state if they are enabled in the STG.

Formally, the composition of specification and implementation can be described as follows. Each state s of the transition system corresponds to a pair of the form $s = \langle c, m \rangle$ where c is a state of the circuit, i.e. an assignment of truth values to all the signals of the circuit, and m is a marking of the STG. The values of the input and output signals defined by c should be consistent with those established by the marking m.

Some of the states of this composition deserve additional comments: the *initial state* and the *error state*. The initial state, s_{in}, is defined by the pair $\langle c_0, m_0 \rangle$ where c_0 defines the initial value of the wires in the circuit and m_0 is the initial marking of the STG. The error state, s_{err}, is a dummy state used to denote incorrect behaviors in the composition.

In a given state $s = \langle c, m \rangle$ of the transition system, the set of enabled events can be defined as follows:

- *Input events*: An input event e is enabled iff a transition labeled with e is enabled in the STG for the marking m.
- *Internal events*: An internal event e is enabled if the corresponding gate is enabled according to the truth values in c.
- *Output events*: An output event e is enabled if the corresponding gate is enabled according to the truth values in c. If there is a transition in the STG labeled with e which is enabled in the marking m, the output event will be *expected* by the environment, otherwise, it will be considered *unexpected*.

An important aspect of this definition is the interaction between internal and environment events: the environment may respond to the outputs of the circuit at any possible time, even if some internal signals of the circuit have not stabilized. In this context, the circuit is said to operate in *input/output mode*. An alternative definition which is not used in this thesis is the *fundamental mode*, where environment events are only considered enabled when no internal events are enabled, i.e. the environment is assumed to be much slower than the circuit. In this thesis, if such timing constraints are required, they will be discovered automatically by the timing analysis.

The transition relation can be characterized very intuitively from the previous definitions. A transition from a state may only fire an event if it is enabled. Transitions may affect the value of the signals, the marking of the STG or both. Also, some of these transitions may denote errors in the interaction between the circuit and the environment.

Formally, given a pair of states, $s_1 = \langle c_1, m_1 \rangle$ and $s_2 = \langle c_2, m_2 \rangle$ and an event e, there is a transition $s_1 \xrightarrow{e} s_2$ in the transition relation if and only if:

- e is an internal event enabled in s_1, $m_1 = m_2$ and c_2 differs from c_1 in the value of the internal signal changed by event e.
- e is an output event enabled in s_1 and unexpected by the environment, and $s_2 = s_{err}$.
- e is an output event enabled in s_1 and expected by the environment, and:
  - c_2 differs from c_1 in the value of the output signal modified by e.
  - m_2 is the marking of the STG reached when e is fired from marking m_1.
- e is an input event enabled in s_1 and
  - c_1 and c_2 differ only in the value of the input signal modified by e.
  - m_2 is the marking of the STG reached when e is fired from marking m_1.

The evolution of the circuit and the environment is defined by a sequence of firings of the underlying transition system, i.e. a *run* of the transition system.

Definition 4.4 (Run of a TS) *Let $A = \langle S, \Sigma, T, s_{in} \rangle$ be a TS. A* run *of A is a sequence $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \dots$ such that $s_1 = s_{in}$ and $s_i \xrightarrow{e_i} s_{i+1} \in T$ for all $i \geq 1$.*

Example 4.4 *Figure 4.4 shows a sample input of the verification problem studied in this chapter. In Fig. 4.4(a), the implementation of the circuit is described as a network of gates. The interaction with the environment is described in the STG in Fig. 4.4(b). Composing the STG with the implementation produces the TS on*

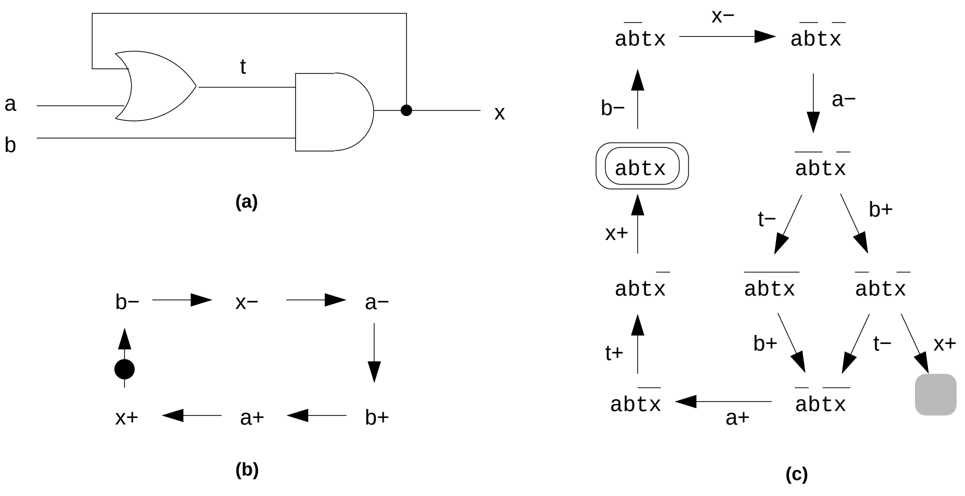


Figure 4.4: Example of the inputs of the verification problem: (a) Implementation of the circuit, (b) STG, (c) Composition as a TS.

Fig. 4.4(c). The enveloped state is the initial state. Note that in this TS there is one transition going to the error state, marked with a shaded area. This transition corresponds to a sequence of firings that is not satisfied by the specification: $x+$ *after* $b+$ *is not allowed in the STG. This* lack of conformance *with the specification can be avoided with timing constraints. For example, if the event* $t-$ *always occurs before* $x+$ *in the state* $\overline{a}bt\overline{x}$*, the circuit behaves conformant to the specification.*

4.2.4 Correctness Criterion

In untimed design styles, the goal is to derive a TS that does not contain any erroneous behavior. Timed circuits accept errors in the TS as long as they are avoided by the temporal characteristics of the circuit and the environment.

One of the elements of our problem is a *correctness criterion* that identifies which states and transitions represent an incorrect behavior. The correctness criterion used in each example depends on the specific problem being studied. Errors in this context include both physical malfunctions and inconsistency with the specifications. Some examples of correctness criteria used in this chapter are the following:

- *Conformance*: Each output produced by the circuit is expected by the environment in the specification, i.e. the output event is enabled in the STG any time that it happens.

- *Hazard freedom*: Any non-input event that becomes enabled must be fired before it becomes disabled, i.e. events do not become disabled when another event is fired. Informally, a hazard occurs when a gate becomes enabled, and then disabled before firing. The output signal of this gate may contain a *glitch*, an unwanted spurious transition which may mislead the gates that are using the value of the signal.

- *Absence of short-circuits*: There should never be a path between “power” and “ground” in the circuit.

In the transition system defined by composing the circuit and environment, some states and transitions may violate the correctness criterion. In order to simplify the analysis, the transition system will be modified in the following way:

- If a state does not satisfy the correctness criterion, it will be replaced by the error state s_{err}.
- If a transition does not satisfy the correctness criterion, we will consider that the target state of the transition is the error state s_{err}.

Thus, timing analysis should discover a set of timing constraints that ensures that the error state is unreachable, i.e. no transition to the error state can be taken. Each transition to the error state represents a different error in the circuit.

Example 4.5 *Let us consider the three circuits depicted in Figure 4.5 together with the STG describing the interaction with the environment. Each circuit describes a different type of error.*

The topmost example depicts a hazard. Signal y may exhibit a glitch as the AND gate has becomes enabled (after $x+$) and disabled (after $t-$) without firing. An element using the value of this signal would observe a spike, which can be interpreted both as a zero or as a one. In the example in the middle, the circuit does not conform with the specification: the circuit does not interact with the environment as it is defined in the STG. An output $z+$ should not occur before an input $y+$. Finally, in the example in the bottom there may be a short-circuit if both transistors are open simultaneously. This situation may occur with the sequence of events depicted on the right.

4.2.5 Delay Model

There are several ways to describe the delay characteristics of a timed circuit, what is called the *delay model*. We will use the following delay model.

Each gate and environment event is assumed to have a non-negligible delay, while wires are assumed to have zero delay, i.e. a *gate* delay model. These delays are considered to be *bounded*. However, the lower and upper bound of these intervals may be defined using a symbolic delay as well as by a constant. The symbolic delay represents an undetermined non-negative delay which is a variable of our problem. Intervals of symbolic delays are often noted as $[d, D]$.

For instance, the delay of a gate can be modeled as $[0, 5]$, $[0, \infty]$, $[2, D_1]$, $[d_2, 4]$ or $[d_3, D_3]$. Each interval should satisfy two basic restrictions: the lower bound

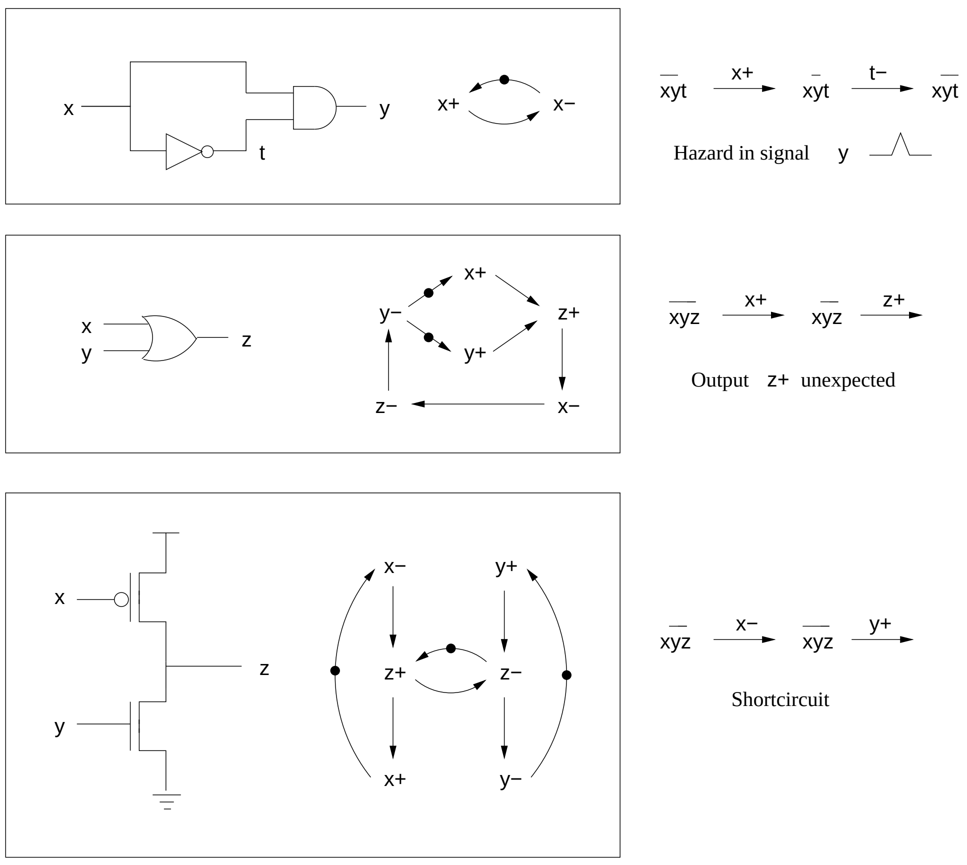

x
y
t
x+
x−
$\overline{x}y\overline{t}$ x+ $xy\overline{t}$ t− $x\overline{y}t$
Hazard in signal y
x
y
z
x+
y−
z+
y+
z−
x−
$\overline{xyz}$ x+ $x\overline{yz}$ z+
Output z+ unexpected
x
z
y
x−
y+
z+
z−
x+
y−
$\overline{x}y\overline{z}$ x− $x y \overline{z}$ y+
Shortcircuit

Figure 4.5: Some classes of errors studied in this chapter: hazards, lack of conformance and short-circuits.

should be greater or equal than 0, and the upper bound should be greater or equal than the lower bound. These restrictions may impose constraints on the symbolic delays that appear in the intervals. For example, the previous intervals impose the constraints $(D_1 \geq 2)$, $(0 \leq d_2 \leq 4)$ and $(0 \leq d_3 \leq D_3)$.

In addition to considering bi-bounded delays, we assume that all delays follow an *inertial* model: any change in the inputs which lasts less than the minimum delay of the gate will be ignored. If the input reverts to the previous value before that time, the output signal will remain unchanged. In contrast, other methods assume an *ideal* or *pure* delay model, where any change in the input signals that enables a gate causes an update of the output signal after $[d, D]$ time units.

These delays are modeling the amount of time *since* an event becomes enabled *until* it fires. Intuitively, there is a clock for each event keeping track of this amount of time. The clock is reset to zero whenever the event is fired or it becomes disabled by other event firings. The delay bounds are establishing limits on the values of this clock: an event e with delay $[d_e, D_e]$ means that e can only be fired when $(d_e \leq clock_e \leq D_e)$. Informally stated, the event cannot be fired before the lower delay bound has elapsed or after the delay bound has elapsed.

A formalism that captures all this information is a timed extension of TS: Timed transition systems (TTS), discussed briefly in Section 3.2.1. TTS annotate each event with a delay interval characterizing the lower and upper delay bound. The original definition of of TTS [92] does not include parameters[2], but an extension to parametric TTS (PTTS) is trivial as follows.

Definition 4.5 (Parametric timed transition system) *A* parametric timed transition system *(PTTS) is a 5-tuple $A = \langle A^-, P, I_P, d, D\rangle$ where $A^- = \langle S, \Sigma, T, s_{in}\rangle$ is a TS called the* underlying transition system*, P is the set of* parameters*, I_P is a satisfiable conjunction of linear inequalities over P called* invariant*, and $d : \Sigma \rightarrow \mathbb{R}^+ \cup P$ and $D : \Sigma \rightarrow \mathbb{R}^+ \cup P \cup \{\infty\}$ respectively associate a* minimal *and a* maximal delay bounds *to each event such that I implies that $\forall e \in \Sigma : (0 \leq d_e \leq D_e)$.*

The definition of PTTS has introduced a new element: an invariant, defined as a set of linear inequalities on the symbolic delays. This invariant describes two types of information. First, it describes information appearing implicitly in the delay bounds of the circuit. For instance, if there is a delay interval of the form $[d, D]$, the constraint $(0 \leq d \leq D)$ will appear in the invariant. Also, these initial invariant can contain information about timing constraints which are known before the verification starts. These constraints can specify design constraints (e.g. *"any input-to-output path in the circuit should be within [20,30]"*) or reasonable assumptions (e.g. *"the environment will not be faster than an inverter"*) that can facilitate the verification process. In all the examples studied in this thesis, the invariant only contains implicit information. No additional information is provided to the algorithm to simplify the analysis.

For the formal definition of the semantics of a PTTS, it is useful to describe a concrete PTTS where each parameter has a specific value. A *concretization* is an assignment of specific values to each parameter of a PTTS such that the assigned values satisfy the invariant. When the temporal behavior of a concrete PTTS is being studied, the symbolic delay bounds can be replaced by the concrete value given by the assignment, while the constant delay bounds remain untouched.

Definition 4.6 (Concretization of a PTTS) *A* concretization *of a PTTS $A = \langle A^-, P, I_P, d, D\rangle$ is a mapping of the form $f : P \rightarrow \mathbb{R}^+$ such that $I_{f(P)}$ is true. The delay bounds assigned to each event in the system can also be concretized as $d^f : \Sigma \rightarrow \mathbb{R}^+$ and $D^f : \Sigma \rightarrow \mathbb{R}^+ \cup \{\infty\}$ such that:*

$$d^f(e) = \begin{cases} f(d_e) & \textit{if } d_e \in P \\ d_e & \textit{otherwise} \end{cases} \qquad D^f(e) = \begin{cases} f(D_e) & \textit{if } D_e \in P \\ D_e & \textit{otherwise} \end{cases}$$

[2]In the remaining of this thesis, the terms "parameter" and "symbolic delay" will be used interchangeably

Definition 4.7 (Timed state sequence [92]) *A* timed state sequence *is a pair $\rho = \langle \sigma, t \rangle$ such that σ is a sequence of states and t is a sequence of time stamps in $\mathbb{R}^+$, $t_1, t_2, t_3, \ldots$ such that $t_1 \leq t_2 \leq t_3 \leq \ldots$ (*monotonic*) and $\forall k \in \mathbb{R}^+ : \exists i \; t_i \geq k$ (*progress*).*

Definition 4.8 (Run of a PTTS) *Let $A = \langle A^-, P, I_P, d, D \rangle$ be a PTTS. A* run *of A is a timed state sequence $\rho = \langle \sigma, t \rangle$ such that σ is a run of the underlying transition system A^- and there exist a concretization $f : P \rightarrow \mathbb{R}^+$ such that:*

- *The lower delay bound is satisfied: $\forall e \in \Sigma, i \geq 0, j \geq i : t_j < t_i + d_e^f : (s_j \xrightarrow{e} s_{j+1} \in \sigma) \rightarrow (e \in \mathcal{E}(s_i))$.*
- *The upper delay bound is satisfied: $\forall e \in \Sigma, i \geq 0 : \exists j \geq i : t_j \leq t_i + D_e^f : e \notin \mathcal{E}(s_i) \vee (s_j \xrightarrow{e} s_{j+1} \in \sigma)$.*

Example 4.6 *Let us recall the circuit defined in Figure 4.4. The implementation composed with the STG produces a TS that characterizes the possible circuit behaviors. Delay bounds can be included in this TS in several ways. In some cases, some delays are known constants that are fixed by the problem under study. When the delay of a component is random or simply uncontrollable, the interval $[0, \infty]$ can be used to establish the delay bounds. For all other events, the lower and upper delay bounds are modeled with a parameter. The transition system with this delay model is described in Figure 4.6(a).*

In this example, we assume that the rising and falling transitions of a gate have the same lower and upper delay bounds, but more complex delay models are also possible. For example, it is possible to assign a different delay to the falling event and the rising event of the same gate. However, the complexity of the verification is dependent on the number of symbolic delays that appear in the PTTS.

The lower and upper delay bounds may forbid some runs which are valid in the underlying TS. Figures 4.6(b-e) capture this effect in the PTTS: several states and transitions which were previously reachable become unreachable when the delays are taken into account. Different concretizations of the symbolic delays have been used in these examples. For example, in Fig. 4.6(b-c) the error transition due to lack of conformance with the specification is avoided. Meanwhile, in Fig. 4.6(d) the error may still appear in some runs of the TS. Finally, in Fig 4.6(e) the error is unavoidable.

4.2.6 Output of the Verification

The goal of the verification is the automatic discovery of linear constraints on the symbolic delays that guarantee the correct operation of the circuit, i.e. the constraints that make the errors unreachable in the PTTS. The *output* of the verification is a set of timing constraints represented as a set linear inequalities over the symbolic delays which is consistent with the invariant. For example, in the PTTS

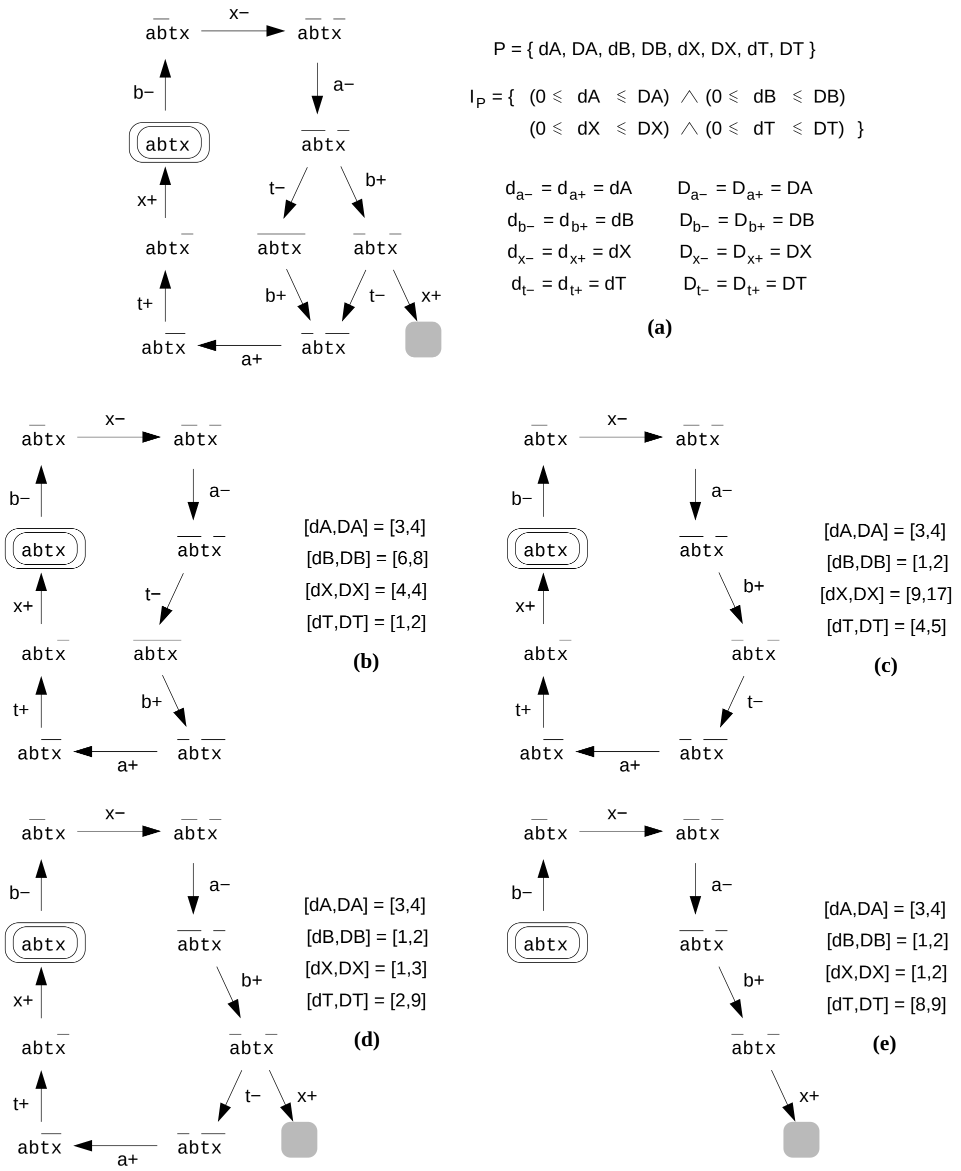

P = { dA, DA, dB, DB, dX, DX, dT, DT }
I_P = { (0 ⩽ dA ⩽ DA) ∧ (0 ⩽ dB ⩽ DB) (0 ⩽ dX ⩽ DX) ∧ (0 ⩽ dT ⩽ DT) }
d_a− = d_a+ = dA D_a− = D_a+ = DA
d_b− = d_b+ = dB D_b− = D_b+ = DB
d_x− = d_x+ = dX D_x− = D_x+ = DX
d_t− = d_t+ = dT D_t− = D_t+ = DT
(a)
[dA,DA] = [3,4]
[dB,DB] = [6,8]
[dX,DX] = [4,4]
[dT,DT] = [1,2]
(b)
[dA,DA] = [3,4]
[dB,DB] = [1,2]
[dX,DX] = [9,17]
[dT,DT] = [4,5]
(c)
[dA,DA] = [3,4]
[dB,DB] = [1,2]
[dX,DX] = [1,3]
[dT,DT] = [2,9]
(d)
[dA,DA] = [3,4]
[dB,DB] = [1,2]
[dX,DX] = [1,2]
[dT,DT] = [8,9]
(e)
abtx
x− a− b+ t− b+ t− x+ a+ t+ x+ b−

Figure 4.6: An example of the effect of delays in a PTTS: (a) TS from Figure 4.4 extended with symbolic delays as a PTTS, (b-e) Reachable state space with different values for the symbolic delays.

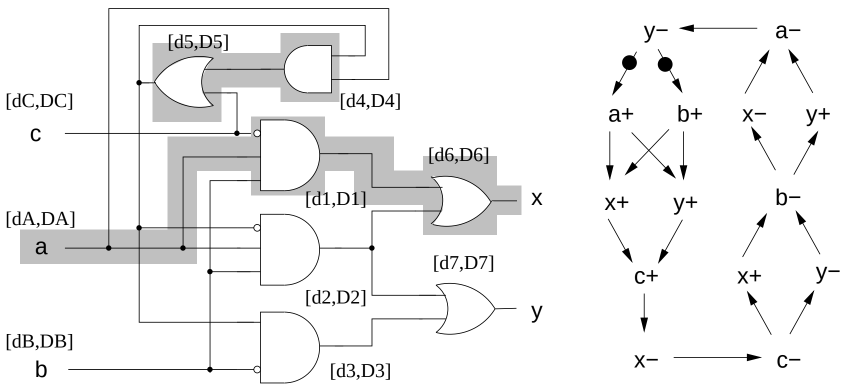
described in Figure 4.6(a), the error can be avoided if the following inequality holds:

$$dB + dX \leq DT$$

Any choice of bounded delays for the gates and environment events which is consistent with this timing constraint will guarantee that the circuit is correct according to the correctness criterion, which in this case is *conformance*.

Another example of verification is shown in Figure 4.7. In this circuit, two timing constraints are required to guarantee conformance and hazard-freedom. Note that each constraint is implicitly comparing the delay of two paths in the circuit: one path should be faster than another path of the circuit. This comparison is highlighted in the Figure 4.7 for the first timing constraint.

If the verification algorithm cannot find a set of timing constraint that is sufficient to guarantee correctness, it will return *false* as the timing constraint. Remember that the analysis using abstract interpretation is approximate but conservative. By conservative we mean that the set of timing constraints provided by the algorithm is always a guarantee of correctness: there are *no false positives*. On the other hand, as the algorithm is approximate, it may not find the least restrictive set of timing constraints. It is also possible that the algorithm fails to find a sufficient set of timing constraints, even if it exists: there can be *false negatives*. However, the experimental results show that false negatives do not occur often enough to be of concern.



$$\boxed{\text{(D4 + D5} < \text{dA + d1 + d6)}} \wedge (D1 < dC + d2 + d7)$$

Figure 4.7: The `nowick` asynchronous controller. Left, the network of gates. Right, the STG describing the interaction with the environment. Bottom, timing constraints required for correctness. The first timing constraint is highlighted as the shaded area in the implementation.

4.3 Computation of Timing Constraints

4.3.1 Overview of the Algorithm

The input to our algorithm is an implementation of a timed circuit as a network of gates, an specification of the interaction with the environment as a STG, a correctness criterion and the description of the delays of each event. The output should be the set of timing constraints that ensure that all errors identified by the correctness criterion are unreachable.

The first step is the computation of the untimed state space as a composition of the circuit (network of gates) with the environment (STG). The result is a PTTS where several transitions and states are labeled as *errors*. Several simplifications are performed on this PTTS before the timing analysis begins.

Timing analysis is the core of our algorithm. In this phase, we assign a clock variable to each event of the transition system. Using abstract interpretation techniques, we track the possible values of clocks and symbolic delays in each state of the system. These possible values are encoded as linear inequalities among clocks and symbolic delays. The timing constraints will be derived from the inequalities that appear in the error transitions.

4.3.2 Computation of the Untimed State Space

In the input, the circuit is described as a network of logic gates. Meanwhile, the interaction with the environment is modeled as a STG. The composition of both elements yields the *untimed state space*, the set of states that are reachable when event delays are not considered. The initial state is defined by the initial values of the signals in the circuit and the initial marking of the STG. The generation of the state space can be performed using a depth-first (DFS) or breadth-first (BFS)traversal starting from that initial state.

In our tool, we have used a simple DFS algorithm to compute the untimed state space. During the DFS traversal, two additional computations are performed: the *detection of cycles* in the TS and the annotation of each state with its number in *reverse DFS postorder*.

Cycle information is required by the timing analysis algorithm, as it is based on abstract interpretation. In Section 2.3.4, it was shown that a special operation called widening must be used in each cycle to guarantee the termination of the analysis. The detection of cycles in a graph during a DFS traversal can be performed as follows: whenever a new state is found, it is marked as *visited*. This mark is removed when all successors of the state have been visited by the DFS traversal. While visiting a state, any transition to a state with the *visited* mark is called a *back edge*. Back edges are the transitions that close a cycle in the TS. Figure 4.8 (a) shows an example of a graph with the set of back edges. Note that different DFS traversals may choose a different set of back edges, but they will always detect one back edge in each cycle.

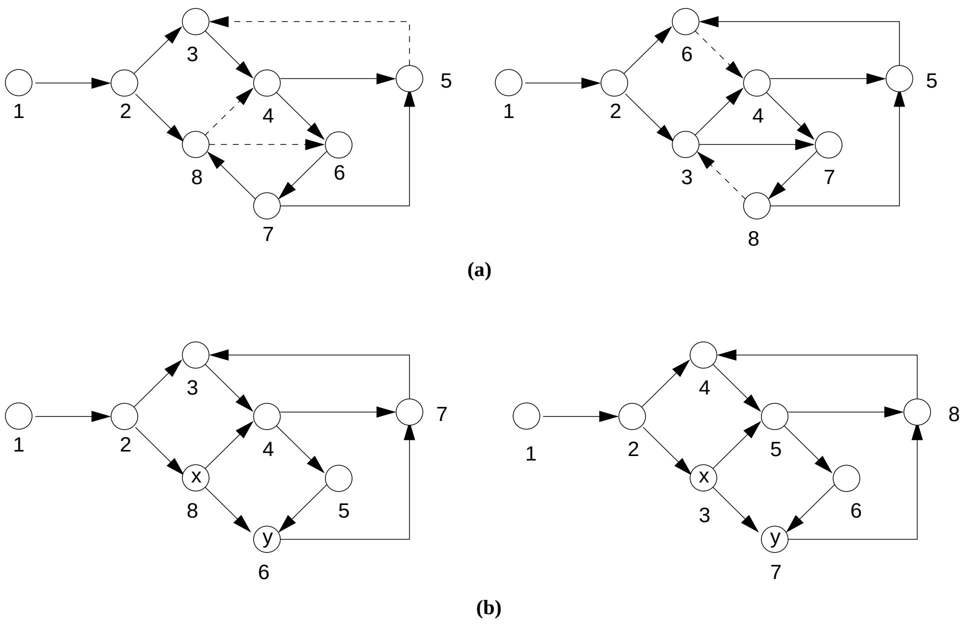


Figure 4.8: (a) Computation of back edges in two different DFS traversals of the same PTTS. Back edges are highlighted with a dashed line. (b) Comparison of the DFS order (left) with the reverse DFS postorder (right).

The computation of the reverse DFS postorder is simple: a number is given to a node when all its successors in DFS order have been numbered. The first node to be numbered gets the highest number n, while the next node to be numbered gets $n - 1, \ldots$ Reverse DFS postorder is used during timing analysis to choose the order of evaluation of the equations. As it was discussed in Section 2.3.3, a good evaluation ordering can accelerate an abstract interpretation analysis. Figure 4.8(b) shows an example of reverse DFS order compared to DFS order. The DFS order correspond to the order in which states are visited in the DFS traversal. With the same order of traversal, reverse DFS postorder achieves an ordering which is much better for abstract interpretation analysis. Intuitively, the reverse DFS postorder seeks to analyze all the predecessors of a node before studying that node, something which propagates changes and avoids recomputations. For instance, using the DFS ordering the equation for node y would be evaluated before the equation for node x. Any change in node x would not be propagated to node y until all the other equations were applied. This does not happen in the reverse DFS postorder, as node x is evaluated before node y.

During the computation of the untimed state space, we will identify several states and transitions that do not satisfy the correctness criterion. We will denote an *error transition* as a transition t of the TS such that (a) t does not satisfy the correctness criterion or (b) the target state of t does not satisfy the correctness criterion.

Algorithm *Untimed_State_Space*(ng, stg, c)
Input: A network of gates ng, a STG stg and a correctness criterion c.
Output: The untimed state space defined by the composition of ng and stg, with the following additional information. Each transition has two labels stating whether it is back edge and whether it is an error transition. Each state is labeled with its number in reverse DFS postorder.

```
init_state := Init_Marking($stg$) $\times$ Init_Signals($ng$)
order := maxint        { Largest integer in the representation }
$\mathcal{R} := \emptyset$
DFS_Traversal( initState )
return $\mathcal{R}$
```

Algorithm *DFS_Traversal*(current)
Input: The state to be visited *current*. Several global variables used by this method are the network of gates ng, the STG stg, the correctness criterion c, the set of reachable states $\mathcal{R}$ and the reverse DFS postorder *order*.
Output: A boolean stating whether this state is an error. Moreover, this method updates the set of reachable states $\mathcal{R}$, the reverse DFS postorder and the *back-edge* label for transitions.

```
if current does not satisfy the criterion $c$ then return true
else if current $\in \mathcal{R}$ then return false
endif
$\mathcal{R}$ := $\mathcal{R} \cup$ current
visited[ current ] := true
{ Compute the set of enabled events }
$\mathcal{E}(\mathrm{current})$ = Enabled_Gates( $ng$ ) $\cup$ Enabled_Transitions( $stg$ )
no_correct_successors := ($\mathcal{E}(\mathrm{current}) \neq \emptyset$)
{ Recursive traversal of successors }
for each enabled event $e \in \mathcal{E}(\mathrm{current})$ do
  next := Fire_Event( current, e )        { $current \xrightarrow{e} next$ }
  if $current \xrightarrow{e} next$ satisfies the criterion $c$ then
    successor_is_error := DFS_Traversal( next )
    no_correct_successors := no_correct_successors $\wedge$ successor_is_error
    back_edge[ $current \xrightarrow{e} next$ ] := visited[ next ] $\wedge$ $\neg$successor_is_error
  else
    back_edge[ $current \xrightarrow{e} next$ ] := false
  endif
endfor
visited[ current ] := false
{ Assign the reverse DFS postorder after recursively visiting the successors }
reverse_DFS_postorder[ current ] := order
order := order - 1
return no_correct_successors
```

Figure 4.9: Pseudocode of the algorithm that computes the untimed state space.

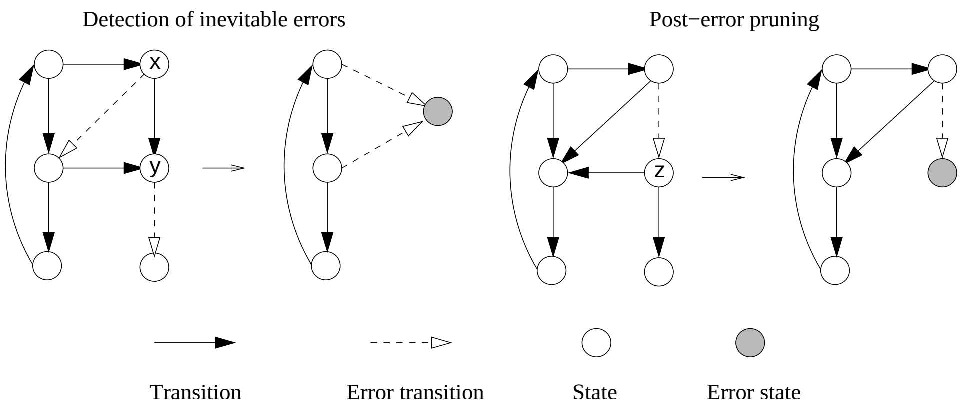


Figure 4.10: Reduction rules for PTTS.

Reductions of the State Space

Abstract interpretation using convex polyhedra has a very high computational complexity, as it was discussed in Section 2.4.6. Therefore, any simplification of the PTTS before timing analysis will reduce significantly the required execution time and memory. The goal of timing analysis is the computation of timing constraints in the errors. As long as the timing constraints for these errors are unaffected, it is possible to eliminate states and transitions of the PTTS that do not contribute to those timing constraints.

For instance, the following reductions have been used:

- *Detection of inevitable errors:* Whenever all the outgoing transitions of a state are errors, mark all the incoming transitions of the state as errors. The meaning of this reduction is that whenever the state is reached, an error is inevitable. Hence, the system should avoid reaching this state as if itself was an error. Due to the high level of concurrency of the circuits under study this is possible if, for instance, an error will occur due to a previous event but there are concurrent enabled events that could be fired before the error.

- *Post-error pruning:* Whenever a transition is only reachable from the initial state through an error, remove the transition.

Example 4.7 *Figure 4.10 shows a graphical example of these reduction rules. On the TS on the left, once states x or y are reached, an error is inevitable. Therefore, any transition leading to x or y can be considered an error. The TS on the right shows that state z is only reachable through an error transition. Therefore, all the transitions and state that can only be reached through z do not need to be considered.*

Additional reduction rules can be defined in order to minimize the TS before the timing analysis. For instance, if reaching an error from a set of states X is impossible, then the set X can be abstracted from the analysis. However, the high concurrency in the circuits under study causes that the pattern described by most reduction rules rarely occurs except in simple examples. The generalization of reduction rules and their potential application will be discussed as future work in Section 6.2.

Explicit vs. Implicit Computation

The algorithm described so far generates the untimed state space explicitly, i.e. each state and transition is represented individually. In contrast, *implicit* or *symbolic* representations encode and manipulate sets of states using decision diagrams. In this way, systems with larger state spaces can be successfully analyzed [100]. However, for this specific problem an implicit representation has several drawbacks.

First, timing analysis is *computationally much more expensive* than the untimed reachability analysis. Therefore, even if an implicit representation can be used to generate the untimed state space in a system where explicit methods fail, it is likely than timing analysis is impractical due to the size of problem.

Also, an implicit representation typically encodes the states of the state space, but it does not represent transitions among states explicitly. Instead, transitions are implicitly defined as a *post-image* (*pre-image*) function that computes the set of successors (predecessors) of a given set of states. This implicit encoding of transitions complicates the analysis with abstract interpretation for several reasons. The first one is the inability to perform some of the reductions described previously. However, the most important factor is the difficulty to detect cycles in the TS. Abstract interpretation needs to apply a special operation called widening in every cycle in order to guarantee termination, as it was explained in Section 2.3.4. Using an implicit representation would lead to either losing the guarantee of termination or sacrificing precision to ensure termination.

Another problem with an implicit representation is the complexity of an implicit timing analysis algorithm. A decision diagram representing states only needs to encode boolean variables. Several classes of diagrams, such as BDDs and ZDDs, can store this type of information. However, each state of the system is labelled with a set of timing constraints defined as linear inequalities on the symbolic delays and clocks. Timing analysis can only be performed implicitly if the linear timing constraints are also encoded in the decision diagram. Few diagrams support this type inequalities, e.g. the DDC and HRD introduced in Section 3.2.5 which have a very high computational complexity. Section 5.3 will also introduce another class of decision diagrams able to encode linear inequalities, but again the reduction in memory usage is traded for a large increase in the execution time of the analysis.

4.3.3 Timing Analysis by Abstract Interpretation

The events of the PTTS can only be fired if their lower and upper bound restrictions are satisfied. Intuitively, each event has an associated event clock that stores the amount of time elapsed since the transition became enabled. Each time an event is fired, event clocks have to be modified accordingly. An analysis of the values of event clocks can reveal whether an event can be fired or not in a given state, and *what values of the symbolic delays* allow the firing. This section presents an algorithm based on abstract interpretation that computes a conservative upper approximation of the values of event clocks.

Timing analysis uses the convex polyhedron abstract domain to capture the timing information. Each state is attached a convex polyhedron with the following variables: the set of event clocks, the set of symbolic delays and a temporary variable called *step*. The *step* variable is used to model the evolution of time. The convex polyhedron attached to a state s is denoted as $\mathsf{Time}(s)$. This abstraction describes the values of clocks when a state is reached, as well as the values of symbolic delays that allow the state to be reached, i.e. the *precondition* of the state.

The system of forward equations in the abstract interpretation framework is easy to define. The locations of interest, as we have said previously, are the preconditions of each state of the PTTS. The equations should capture how time elapses as the events are fired. hen an state is reached, several events become enabled while other events that were enabled previously continue to be enabled. These events have to be fired according to its lower and upper delay bound, taking into account that some events have already been enabled for some time. We have defined a symbolic function called *transfer* (explained in detail the following section) that advances the clock values while satisfying all upper and lower bounds. The output of this function is the value of clocks after firing an event, i.e. the *postcondition* of the transition being taken. Using this function, the abstractions for states can be defined as the following system of equations:

$$\forall m \in S, n \xrightarrow{e} m \in T \;:\; \mathsf{Time}(m) = \bigcup \mathrm{transfer}(n, e, m)$$

This definition has been simplified to simplify the presentation, as, for instance, the widening operator should be used for the incoming transitions that are back edges.

Example 4.8 *For example, let us consider the circuit defined previously in Figure 4.4. If* $\mathsf{Time}(S)$ *denote the set of possible clock valuations in a state* S *and (*$\mathsf{Time}(S) \xrightarrow{x}$*) denotes the possible clock valuations after firing an event* x *from an state* S*, the following system of equations can be defined.*

```
Algorithm AbstractInterpretation ($R$)
Input: A parametric timed transition system $R = \langle\langle S, \Sigma, T, s_{in}\rangle, P, I_P, d, D\rangle$.
Output: The abstraction Time for all states.

  for each state $s \in S$ do  Time($s$) := $\emptyset$  endfor
  Time($s_{in}$) := $I_P \ \wedge\ (\ \forall\, e\ \in\ \mathcal{E}(s_{in}) : \{\ clock_e\ =\ 0\ \}\ )$
  $changed$ := $\{s_{in}\}$
  do
    $n$ := state in $changed$ with lowest reverse DFS postorder number
    $changed$ := $changed \setminus \{n\}$
    for each transition $n \xrightarrow{e} m \in T$ do
      $newTime$ := $transfer(n, e, m)$
      if ($newTime \subseteq$ Time($m$)) then continue endif
      $newTime$ := $newTime\ \cup$ Time($m$)
      if back_edge[ $n \xrightarrow{e} m$ ] then
        Time($m$) := (Time($m$) $\nabla$ $newTime$) $\cap\ I_P$
      else
        Time($m$) := $newTime\ \cap\ I_P$
      endif
      $changed$ := $changed\ \cup\ \{m\}$
  while ($changed \neq \emptyset$)
```

Figure 4.11: Abstract interpretation algorithm

$$\mathsf{Time}(\mathtt{abtx}) = InitValues\ \cup\ (\mathsf{Time}(\mathtt{abt\overline{x}}) \xrightarrow{\mathtt{x+}})$$
$$\mathsf{Time}(\mathtt{\overline{a}b\overline{tx}}) = (\mathsf{Time}(\mathtt{\overline{abtx}}) \xrightarrow{\mathtt{b+}})\ \cup\ (\mathsf{Time}(\mathtt{\overline{a}bt\overline{x}}) \xrightarrow{\mathtt{t-}})$$
$$\mathsf{Time}(\mathtt{a\overline{b}tx}) = (\mathsf{Time}(\mathtt{abtx}) \xrightarrow{\mathtt{b-}}) \qquad \mathsf{Time}(\mathtt{a\overline{b}t\overline{x}}) = (\mathsf{Time}(\mathtt{a\overline{b}tx}) \xrightarrow{\mathtt{x-}})$$
$$\mathsf{Time}(\mathtt{\overline{ab}t\overline{x}}) = (\mathsf{Time}(\mathtt{a\overline{b}t\overline{x}}) \xrightarrow{\mathtt{a-}}) \qquad \mathsf{Time}(\mathtt{\overline{abtx}}) = (\mathsf{Time}(\mathtt{\overline{ab}t\overline{x}}) \xrightarrow{\mathtt{t-}})$$
$$\mathsf{Time}(\mathtt{\overline{a}bt\overline{x}}) = (\mathsf{Time}(\mathtt{\overline{ab}t\overline{x}}) \xrightarrow{\mathtt{b+}}) \qquad \mathsf{Time}(\mathtt{ab\overline{tx}}) = (\mathsf{Time}(\mathtt{\overline{a}b\overline{tx}}) \xrightarrow{\mathtt{a+}})$$
$$\mathsf{Time}(\mathtt{abt\overline{x}}) = (\mathsf{Time}(\mathtt{ab\overline{tx}}) \xrightarrow{\mathtt{t+}})$$

Intuitively, each equation defines the clock values in a state as the union of clock values after its incoming transitions. In the initial state, the enabled clocks are set to zero, while the delays can have any value that satisfies invariant. In the example from Figure 4.4 the values for clocks and delays in the initial state are:

$$\begin{aligned} InitValues &= \{\ InitClocks\ \wedge\ Invariant\ \} \\ InitClocks &= \{\ clock_{b-} = 0\ \} \\ Invariant &= \{\ (0 \le d_{a+} \le D_{a+}) \wedge (0 \le d_{a-} \le D_{a-}) \wedge \\ &\quad (0 \le d_{b+} \le D_{b+}) \wedge (0 \le d_{b-} \le D_{b-}) \wedge \\ &\quad (0 \le d_{AND} \le D_{AND}) \wedge (0 \le d_{OR} \le D_{OR})\ \} \end{aligned}$$

Figure 4.11 describes an algorithm that computes a solution for the system of equations using a *increasing* fixpoint. Each location except the initial one starts with an empty set of valid assignments to clocks and values, i.e. an empty abstraction. Meanwhile, the initial location begins the analysis with the invariant, and the clocks of all enabled events set to zero. The algorithm applies the equations iteratively as long as they add new valid assignments. The solution is reached when there is a fixpoint, i.e.applying all equations another time does not yield new assignments in any location.

4.3.4 Propagation of Clock Values

The core of the analysis is the *clock transfer* function that computes *symbolically* the changes in clock values after firing an event. Clock values are represented by a convex polyhedron, with one dimension per event clock and one dimension per symbolic delay. The restrictions of this polyhedron represent the restrictions on the clock values in a given state. Intuitively, the purpose of the transfer function is to make sure that whenever an event e is fired, its delay bounds d_e and D_e are taken into account and added to the restrictions on the clock values.

Event clocks for enabled events store the amount of time elapsed since the event became enabled, while disabled clocks are undefined. After firing an event, event clocks should be updated to reflect the time elapsed between the firing of the last event to the firing of current event. This time spent in the state is called clock *step*, and it should satisfy the following properties:

- Step should be ≥ 0, i.e. no negative time increments
- If the event being fired is x, then its lower and upper delay bounds should be fulfilled: $(d_x \leq clock_x + step \leq D_x)$.
- The upper delay bound of the other enabled events should not be exceeded: ($\forall y$: y is enabled : $clock_y + step \leq D_y$).

When an event e is fired, the clocks of other events have to be updated. The change in their clocks depends on whether they are enabled or disabled before and after firing e. Events that become newly enabled have their clock reset to zero, while events that become disabled have their clock undefined. If an event remains enabled before and after e, its clock is increased by the clock step. Finally, if an event remains disabled, its clock does not change.

Fig.4.12 describes the algorithm that computes the transfer function using convex polyhedra operators. Fig.4.13 shows an example of the computation that would be performed by the algorithm. Events that are enabled before and after firing event e have been increased by an amount in the interval $[d_e, D_e]$, i.e. the unknown clock step. Also, notice that some constraints among the symbolic delays of different events have been discovered. These constraints were imposed over the clock step during the transfer, and *implied* several restrictions on the delays that are made explicit when variable step is undefined. For example, the restriction $D_a \geq d_e$ means that event e can be fired only if a is not faster than e. Otherwise, the postcondition of this transition would be empty, i.e. no assignment to clock and symbolic delays is consistent with the firing of the event. This restriction is implied by the constraints $clock_a + step \leq D_a, clock_e + step \geq d_e$, $clock_a = 0$, $clock_e = 0$.

Algorithm *transfer*(src, e, dst)
Input: An event $src \xrightarrow{e} dst$.
Output: The postcondition of $src \xrightarrow{e} dst$.

```
$P$ := $\mathsf{Time}(src)$
$P$ := $P \wedge (step \geq 0)$
$P$ := $P \wedge (clock_e + step \geq d_e)$
$P$ := $P \wedge (clock_e + step \leq D_e)$
for each event $e' \neq e$: $e' \in \mathcal{E}(src)$ do                      $P := P \wedge (clock_{e'} + step \leq D_{e'})$
for each event $e' \neq e$: $e' \in \{\mathcal{E}(src) \cap \mathcal{E}(dst)\}$ do       $P[clock_{e'} := clock_{e'} + step]$
for each event $e' \neq e$: $e' \in \mathcal{E}(dst) \wedge e' \notin \mathcal{E}(src)$ do  $P[clock_{e'} := 0]$
for each event $e' \neq e$: $e' \in \mathcal{E}(src) \wedge e' \notin \mathcal{E}(dst)$ do  $P[\ abstract\ clock_{e'}\ ]$
if $e \in \mathcal{E}(dst)$ then $P[clock_e := 0]$
else                    $P[\ abstract\ clock_e\ ]$
endif
$P[\ abstract\ step\ ]$
return $P$
```

Figure 4.12: Clock transfer function

4.4 Choice of Timing Constraints

The goal of timing analysis is the computation of timing constraints on the symbolic delays that guarantee the correct operation of the timed circuit. However, timing analysis computes the complementary constraints: the restrictions required to reach each state and transition of the state space. Furthermore, these invariants computed by abstract interpretation may include clock variables, that are irrelevant as we are trying to characterize correctness in terms of the symbolic delays.

The first step towards choosing timing constraints is focusing on the error transitions. In each error transition we seek, only the constraints among symbolic delays. Thus, we will existentially abstract all the clock variables from the convex polyhedra using Fourier-Motzkin elimination (see the subsection on convex polyhedra within Section 2.4.6). For instance, let us consider that the following polyhedron represents the postcondition of an error transitions:

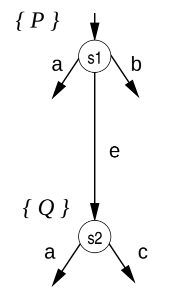
$$(d_a \leq clock_a \leq D_c + D_f) \;\wedge\; (clock_b \leq D_b) \;\wedge\; (clock_a = clock_b)$$

After removing the clock variables, the following constraints on symbolic delays remain:

$$(d_a \leq D_c + D_f) \;\wedge\; (d_a \leq D_b)$$

The timing constraints that avoid the error transitions are the complement of these inequalities. In our example,

$$(d_a > D_c + D_f) \;\vee (d_a > D_b)$$



```
transfer( s1, e , s2 ) =
  P := Time (s1)
  P := P /\ (step >=0)
  P := P /\ ( clock_e + step  >= d_e )
  P := P /\ ( clock_e + step  <= D_e )
  P := P /\ ( clock_a + step  <= D_a )
  P := P /\ ( clock_b + step  <= D_b )
  P [ clock_a := clock_a + step ]
  P [ abstract clock_e ]
  P [ clock_b := ? ]
  P [ clock_c := 0 ]
  P [ abstract step ]
```

$$\{P\} = \{(clock_e = 0) \wedge (clock_a = 0) \wedge (0 \leq clock_b \leq 1)\}$$
$$\{Q\} = \{(clock_c = 0) \wedge (D_e \geq clock_a \geq d_e) \wedge$$
$$(D_a \geq clock_a \geq d_e) \wedge (d_e + 1 \leq D_b)\}$$

Figure 4.13: Example of the transfer function for an event e, with the postcondition Q obtained from a precondition P.

Therefore, the output of the verification will be a conjunction of formulas, one for each error transition. Each subformula will be a disjunction of strict linear inequalities representing all the possible restrictions on the symbolic delays that avoid the error.

$$\underbrace{((\neg ineq1) \ \vee \ (\neg ineq2))}_{error1} \ \wedge \ \underbrace{((\neg ineq3) \ \vee \ (\neg ineq4) \ \vee \ (\neg ineq5))}_{error2}$$

For some applications, this formula can be used directly as the timing constraint. For instance, let us assume that the goal is checking whether a set of known bounded delays satisfies the timing constraints. This check can be done directly using this formula: replace each symbol by the given constant value and check if the timing constraint is satisfied.

However, this formula is complex because it contains disjunctions. Other applications may require that the timing constraint is simplified so that it is presented as a *conjunction* of linear inequalities. This type of formula is much more informative as it highlights the potential races in the circuit explicitly. Also, it opens the possibility to answer efficiently question like *"which delays should be chosen in order to minimize the delay of a path σ while ensuring correctness?"*. Techniques such as linear programming or constraint satisfaction can be employed to solve these queries, using the timing constraints computed by the timing analysis as a guarantee of correctness.

The simplification of the output formula is not trivial. The formula consists of a conjunction of several disjunctions of inequalities. In the simplified result, at least one inequality of each disjunction should be satisfied. A brute force strategy such as computing the disjunctive normal form (DNF) of the formula is not an option, due to the exponential growth in the size. Simplifying the formula with a Presburger arithmetic tool like Omega [130] is again too inefficient except in the smallest examples. A more practical strategy would be *selecting one inequality* from each disjunction. This selection should attempt to produce a conjunction of inequalities which is (a) *non-contradictory*, (b) *compatible with the invariant* satisfied by symbolic delays and (c) the *least restrictive possible*. Again, simply trying all possible combinations is not feasible as there are too many of them.

The simplification of the formula has been fully automated using a backtracking algorithm guided with several heuristics. This algorithm proceeds by selecting, one inequality at a time, the most promising inequality among all candidates to timing constraints appearing in the formula. At all times, the current set of timing constraints should be both satisfiable and compatible with the invariant, otherwise the procedure should backtrack and reconsider the last choice. An error transition may be avoided by the current selection of timing constraints because an inequality of the disjunction appears in the selection. However, it may also be avoided if the current selection *implies* an inequality from the disjunction, e.g. the timing constraints $(d_a > D_b + D_c) \wedge (D_b > D_d)$ imply the inequality $(d_a > D_c + D_d)$. If the error transition is avoided by the current choice of timing constraints, the constraints from its disjunction are no longer considered as candidates.

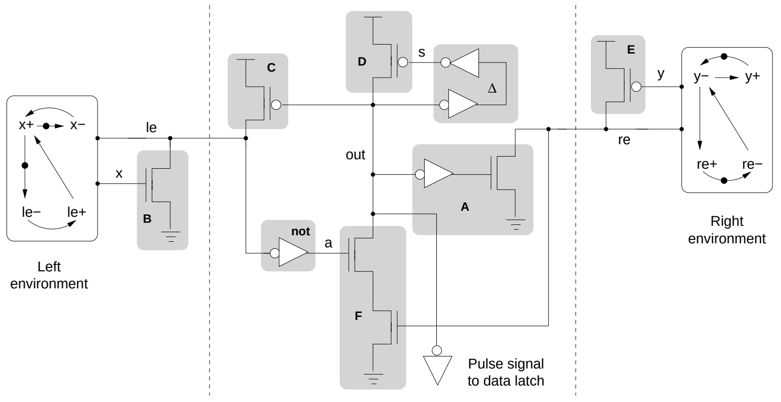
Several heuristics are used to select the best timing constraints among the candidates. These heuristics favor the following kinds of constraints:

- Constraints where a long sequence of delays must be slower than a much shorter path, e.g. $(d_x + d_y + d_z > D_t)$.
- Constraints where environment delays must be slower than a path inside the circuit, e.g. $(d_{b+} > d_{NOT} + d_{AND})$.
- Constraints that appear in several failure transitions of the circuit. Due to the concurrency in the circuit, a single error might be the cause of different failure transitions. For instance, if a transition where *“a+ happens before b+”* is an error, there can be several failure transitions derived from this single conceptual error. Thus, a constraint that avoids several failures is preferred to constraints that avoid a single failure.

Currently, the timing constraints are selected automatically by a backtracking procedure based on these heuristics. This procedure computes the best k sets of consistent timing constraints according to the heuristics. Computing all the possible combinations would also be possible but very inefficient. This procedure is executed *after* the timing analysis, and it does not require repeating the timing analysis phase. In contrast, some related approaches [158] select one timing constraint at a time and repeat the timing analysis phase to detect new timing constraints.

4.5 Experimental Results

We have implemented the algorithms presented in this chapter in a verification tool. For the convex polyhedron abstract domain, we have used the the New Polka



$$\begin{array}{rcl}
d_{y-} + d_A & > & D_D + D_\Delta \\
d_{x+} + d_B + d_F & > & D_{y-} + D_{y+} + D_A \\
d_{x+} & > & D_\Delta + D_D + D_{not} \\
d_{not} + d_B + d_F & > & D_{x-} \\
d_D + 2d_\Delta & > & D_{y-} + D_A + D_E \\
d_\Delta & > & D_{not} + D_C \\
d_D + d_\Delta & > & D_A \\
d_{x+} + d_B & > & D_D + 2D_\Delta \\
d_{y+} & > & D_E \\
d_{x-} & > & D_B
\end{array}$$

Figure 4.14: GasP FIFO controller. Each shaded area has been modeled with a different symbolic delay. On the bottom, the discovered timing constraints that are sufficient to guarantee the correct operation of the circuit.

convex polyhedra library [99], implementing integers as fixed precision `long` integers. In this section, we show some examples that have been verified using this tool, from the domain of asynchronous controllers. These experimental results have been collected on a Pentium 4 2Ghz machine with 512Mb RAM running Linux. CPU time and memory have been measured using the utility memtime [22].

4.5.1 GasP FIFO Controller

We have formally verified a GasP FIFO controller from Sun Microsystems [149]. This circuit handles the flow of data between stages of a pipeline: whenever the previous stage is FULL and the next stage is EMPTY, the control circuit (a) produces a pulse to the data latch in order to make it transparent, (b) declares that the next stage is FULL and (c) declares that the previous stage is EMPTY. The state of a stage is encoded in a single wire, where EMPTY (FULL) is encoded as HI (LO). Fig.4.14 shows the controller of one stage of a pipeline. The environment of this controller corresponds to the previous and next stages of the pipeline. Notice that wire le corresponds to the wire re in the previous stage of the pipeline, i.e. the signal le is both an input and an output in this type of circuit.

This asynchronous controller is designed to achieve a very high throughput, so it depends on timing constraints for its correct operation. In [102], this circuit is verified and sufficient relative timing constraints to ensure correctness are derived. The GasP circuit has also been verified using generalized relative timing [142]. Furthermore, pipelines based on GasP units have been verified hierarchically using chain timing constraints [103]. However, it is hard to translate these constraints into restrictions on the gate and transistor delays.

The correctness of the circuit has been verified with respect to three criteria: *absence of short-circuits*, *absence of hazards*, and *conformance*. These criteria can be satisfied with the timing constraints that appear in Fig.4.14.

4.5.2 Asynchronous Pipeline

We have also verified an asynchronous pipeline with different number of stages and an environment running at a fixed frequency. The processing time required by each stage has different min and max symbolic delays. The safety property being verified in this case was *"the environment will never have to wait before sending new data to the pipeline"*. Fig.4.15 shows the pipeline, with an example of a correct and incorrect behavior. The tool discovered that correct behavior can be ensured if the following holds:

$$d_{IN} > D_1 \ \wedge \ \ldots \ \wedge \ d_{IN} > D_N \ \wedge \ d_{IN} > D_{OUT}$$

where D_i is the delay of stage i, and d_{IN} and D_{OUT} refer to environment delays. This property is equivalent to:

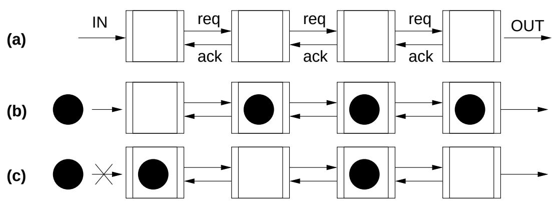
$$d_{IN} > max(D_1, \ldots, D_N, D_{OUT})$$

Therefore, the pipeline is correct if the environment is slower than the slowest stage of the pipeline.

This pipeline example is interesting due to the high degree of concurrency that it exhibits. Increasing the length of the pipeline by one stage makes the verification problem more complex, adding two clock variables, two new symbolic delays, and multiplying the size of the state space by a factor of 3. CPU time for pipelines of a different length can be found in Fig.4.15.

4.5.3 Other Examples

Several asynchronous controllers from the literature have also been verified with our timing analysis algorithm. Some of these control circuits have been previously



# of stages	PTTS States	PTTS Trans	# of symbols	CPU Time (seconds)
2	36	88	8	0.6
3	108	312	10	2
4	324	1080	12	13.5
5	972	3672	14	259.2

Figure 4.15: (a) Asynchronous pipeline with N=4 stages, (b) correct behavior of the pipeline and (c) incorrect behavior. Dots represent data elements. On the bottom, the CPU times required to verify pipelines with different number of stages.

Table 4.1: Experimental results

Example	Circuit		STG		PTTS		Sym	TC	CPU
	Wires	Gates	Places	Trans	States	Trans			
nowick	10	7	19	14	60	119	10	2	0.5s
gasp-fifo	9	7	10	8	66	209	12	10	8.1s
sbuf-read-ctl	13	10	19	16	74	157	14	4	1.2s
rcv-setup	9	6	14	15	72	187	12	8	2.1s
alloc-outbound	15	11	21	22	82	161	19	3	1.3s
ebergen	11	9	16	14	83	188	13	5	1.3s
D flip-flop	6	4	16	22	146	448	8	7	5.8s
mp-fwd-pkt	13	10	24	16	194	574	12	6	1.9s
chu133	12	9	17	14	288	1082	7	3	1.3s
converta	14	12	16	14	396	1341	14	13	20.4s

verified in [133] using a Relative Timing approach. In these circuits, correctness has been defined as *absence of hazards* and conformance *conformance*. Table 4.1 shows shows the size of the circuits, STGs and the computed PTTS, the number of parameters used as symbolic delays (Sym), the number of linear inequality timing constraints required for correctness TC, and the CPU time (in seconds) used for the verification. Intuitively, the number of timing constraints required for the correctness of a circuit can be interpreted as a measure of its internal complexity. In general, a circuit which is mostly combinatorial will require less timing constraints than a circuit with many feedback loops and internal race conditions.

4.5.4 Evaluation of the Results

The timing analysis algorithm uses the convex polyhedron abstract domain, which has an exponential complexity with respect to the number of variables. Therefore,

Table 4.2: Quantifying the relevance of symbolic delays.

# of	TTS		All stages symbolic			Only 1 stage symbolic		
stages	States	Trans	Sym	CPU	Mem	Sym	CPU	Mem
2	36	88	8	0.6s	64Mb	4	0.5s	62Mb
3	108	312	10	2s	67Mb	4	1.9s	65Mb
4	324	1080	12	13.5s	79Mb	4	3.3s	70Mb
5	972	3672	14	259.2s	147Mb	4	9.8s	102Mb
6	2916	12312	16	O/M	O/M	4	43.6s	187Mb

O/M = out of memory ($>$1.5Gb)

the number of symbolic delays in a circuit plays an important role in the overall complexity of the method. Table 4.2 shows some data that quantify the importance of symbolic delays in the asynchronous pipeline example. Two instances of the problem have been verified: one where each state of the pipeline has symbolic lower and upper delay bounds, and another where all stages except one have constant delay bounds. The number of symbolic delays in each case is described in the column Sym. The results show that defining less symbolic delays allows the study of much larger systems: the complexity is not significantly affected by the size of the state space, when compared to the effect of the number of symbolic delays. Therefore, it is convenient to minimize the amount of parametric delays to ensure an efficient analysis. On a related note, most verification algorithms for parametric timed systems can only deal with a small number of parameters (for instance < 10 or < 4, see Sections 3.2.5 and 4.6 for specific details). Therefore, the analysis of circuits with up to 20 symbolic delays reflects an improvement with respect to these methods.

The examples that could not be verified failed due to running out of memory. More precisely, the part of the algorithm where it happens is the procedure that computes the dual representations of a convex polyhedron. As we discussed in Section 2.4.6, the conversion among dual representations can increase the size of the polyhedron exponentially. A detailed inspection of our examples has shown that some polyhedra have a large number of inequalities, because in each state we keep track of a large number of competing paths in the circuit. For instance, in all the examples, the average number of constraints in each polyhedron is between 13 and 27, but some polyhedra have more than 50 inequalities. This factor, combined with the possibility of exponential blowup in every conversion, limits the scalability of the algorithm.

Another aspect of these results should be mentioned. In many applications of convex polyhedra, a large amount of time is spent simplifying the coefficients of the linear inequalities. Instead of working with rationals, the coefficients are simplified to become integers. For instance, the following inequality with rational coefficient

$$\frac{2}{3}x \; - \; \frac{4}{7}y \; \geq \; \frac{5}{2}$$

can be simplified to use only integer coefficients as $(28x - 24y \geq 105)$. This simplification requires the computation of the greatest common divisors among several coefficients and performing several integer products and divisions. Furthermore, in some domains these computations increase the magnitude of the integer coefficients in the inequalities, forcing the use of arbitrary precision integers instead of machine fixed precision integers to encode the coefficients. Using arbitrary precision adds a large time and memory overhead. Nevertheless, this does not happen in our examples as most coefficients of the inequalities are unit, i.e. either 0 or ± 1. To quantify this observation consider that in all the examples, the percentage of inequalities that contain *only* unit coefficient is between $77 - 99\%$. If we exclude the GasP FIFO example, a circuit with a very complex internal behavior which creates a large number of invariants, this percentage grows to $95 - 99\%$. The abundance of unit coefficients is explained because most inequalities compare the delay of two paths in the circuit, and these paths do not tend to contain cycles, therefore each symbolic delay appears at most once.

4.6 Related Work

There are several previous methods which have incorporated symbolic delays in the verification of a timed circuit. After describing our methodology in detail, we are in condition to summarize these methods and discuss their relationship with this thesis.

Symbolic Timing Verification

The term *symbolic timing verification* has been used with two different meanings in the literature. In some papers, symbolic refers to the use of representations based on decision diagrams, e.g. [34, 95], even though the system is described with metric timing constraints. However, in [9, 87], symbolic timing verification is used to denote the verification of a timed system with symbolic delays.

In [9], an approach for symbolic timing verification based on constraint programming is presented, even though the class of circuits that can be analyzed is very restricted, e.g. no feedback loops. Also, the tool MTV [87] performs symbolic timing verification of microprocessor-based designs. Note that these designs are not asynchronous, as there is a clock signal. This tool can handle timing constraints that span through more than one clock cycle (multi-cycle constraints). However, it suffers from the same weakness as the previous technique: it cannot handle feedback loops without latches. Therefore, it cannot be used to analyze sequential asynchronous circuits.

Time-Symbolic Simulation

An approach which is closely related to the symbolic delay verification presented in this thesis is *time-symbolic simulation* [98]. Time-symbolic simulation deals with the verification of combinatorial asynchronous circuits, modeling each gate and environment delay using a single symbolic variable. The verification enumerates all possible event traces, recording the delay assumptions for each trace as a set of linear inequalities. The timing constraints produced by this method are very similar to those computed with our method.

However, time-symbolic simulations shows several shortcomings with respect to the work presented in this thesis. First, *it can only deal with combinatorial circuits*: there cannot be feedback loops in the circuit. This assumption is not reasonable in the domain of asynchronous controllers. For instance, all the circuits that appear in this thesis are sequential circuits that contain loops. Moreover, time-symbolic simulation *only deals with a single input pattern at a time*, verifying the expected output and the absence of hazards. As a result, it cannot accurately verify complex environments specified with a STG. In a STG, sequences of inputs are possible, and the timing information carried from the previous input can influence the evaluation of the next input. Obviously, encoding all the input traces represented in a STG is not practical so this method is limited to simple environments. Finally, delays are modeled using a *fixed delay model*, i.e. the delay of a gate is modeled as a single symbol. The fixed delay model is usually not realistic, as the delay of a gate can be altered slightly by factors like temperature, power supply, . . . Representing bounded delays is possible, but it requires using a different variable for each occurrence of the same event, something which again limits the applicability of the technique. To sum up, time-symbolic simulation uses less realistic assumptions than those used in this thesis.

Coded-Time Symbolic Simulation

Another related approach is *coded-time symbolic simulation* [129]. The concept behind this method is the following: each delay is assumed to be fixed, but only a lower and upper bound of its value is known. A discrete notion of time is used, so the number of possible values within this interval is finite and can be enumerated. Again, the method works by enumerating all event traces and recording the delay of each trace. The possible values of this delay are encoded using a BDD, as the values of each delay can be enumerated. Contrary to time-symbolic simulation, this method can be used even when the circuit has feedback loops. Still, this method cannot generate the powerful linear symbolic constraints achieved with the previous approach.

The are two main drawbacks to this method. First, the “uncertain but constant” delay model is again less realistic than the bounded delay model. Also, the efficiency of the method depends not only on the size of the circuit, but in the range of the delay intervals and the size of the discrete time units: intervals with more values require more boolean variables to encode the delay. Consequently, the work presented in this thesis uses a more realistic delay model, provides more meaningful constraints and is more efficient in general than coded-time symbolic simulation.

Symbolic Time Separation of Events

The problem of time separation of events, studied with metric timing constraints, can also be analyzed in a circuit with bounded symbolic delays. In [11, 95], an algorithm to compute the symbolic time separation between events is presented. The delays in the circuit are modeled as bounded symbolic delays, and the time separation is represented as min/max expressions in Presburger arithmetics. This technique has a very high computational complexity which severely limits the number of symbolic delays that can be used in the circuit (at most 4 symbolic delays appear in the examples). Finally, even though this approach uses bounded symbolic delays, it does not address the problem of verification of a timed circuit.

Symbolic Verification of Timing Diagrams

A timing diagram is a graphical representation of the waveforms of the signals that can describe the relative order among events in a circuit. Some of these relationships define properties enforced by the environment while others establish necessary constraints for correctness. In [10], the verification of these necessary condition is addressed. The expressiveness of these diagrams is extended by adding can be increased by adding bounded delay intervals that describe the time elapsing between different events in the diagram. Also, these intervals can be defined with symbolic delays. The result of the verification is a quantifier-free Presburger arithmetic formula which describes the necessary constraints that ensure a correct operation. This formula contains linear inequalities among the symbolic delays that, in addition to intersections, can be combined using unions. In this sense, the constraints generated by this approach are more general than the timing constraints studied in this thesis. However, the size of the timing diagrams is much smaller than the asynchronous controllers. In addition, similar to the symbolic time separation of events, the computational complexity of this approach is very dependent on the number of symbolic delays. At most, 12 symbolic delays are used in the examples.

Verification Based on Parametric Timed Automata

Previous work proved that the verification of a timed circuit can be expressed as an equivalent verification of a timed automata [108]. Also, methods for the verification of parametric timed automated have been described (see Section 3.2.5). Combining these two contributions, the verification of timed circuits with symbolic delays using parametric timed automata was introduced in [43]. This contribution appeared after the approach presented in thesis was described [48, 49, 52]. No experimental results are provided, and the example circuits are very small. Given the high complexity of the methods dealing with parametric timed automata [97], it is unlikely that they this method will be able to scale to the size of circuits analyzable in this thesis.

Derivation of Constraints by Failure Analysis

A hybrid approach between metric timing and symbolic delays is presented in [158]. The problem to be solved is the verification of a timed circuit with symbolic delays. Instead of using symbolic delays during the analysis, this method uses Integer Linear Programming (ILP) to select sufficiently large constant intervals. The analysis of the circuit is performed using metric timing techniques until an erroneous trace is found. Then, the analysis discovers and chooses a linear inequality among the symbolic delays that is sufficient to avoid that trace. Another round of ILP selects new intervals of constant bounds for the delays that also satisfy this new constraint. When the analysis cannot find an erroneous trace, the linear inequalities and the constant bounds used in the last metric timing analysis pass are the constraints required for correctness.

A weakness of this technique is the necessity to include these constant bounds in the set of timing constraints. After all, there might be an erroneous trace that satisfies all the linear inequality constraints and is only avoided by the specific constant bounds select by ILP. In contrast, the approach presented in this thesis finds a set of sufficient linear inequality constraints: no additional metric timing constraints are available.

Summary

There have been several attempts to apply the techniques from parametric timed systems in the analysis of circuits with symbolic delays. However, these attempts exhibit several weaknesses that limit their applicability: lack of full automation, extremely high computational complexity, problems to deal with a bounded delay model, dependence on metric timing constraints at some level or inability to deal with sequential (cyclic) circuits. These problems have been addressed by the methodology presented in this chapter.

4.7 Conclusions

An automatic method for the verification of timed circuits has been presented. The output of the algorithm is a conservative approximation of the values of clocks and symbolic delays in the reachable states of the system. An application has been shown by computing the constraints of gate and input delays in an asynchronous circuit that guarantee correct behavior. Remarkably, the approach works for more than 15 symbolic delays within a reasonable time, significantly better than alternative methods presented in Section 4.6.

The proposed methodology studies the circuit without specifying delays for each component, even though known delays can be used in the analysis. The timing constraints provided by this methods are a system of linear inequalities over the symbolic delays. The proposed methodology offers several advantages with respect to the alternative methods for the verification of timed systems presented in Section 3.3, achieved at the cost of higher computational complexity. Some of these advantages are the following:

- The discovered timing constraints are independent of the technology and the delays of the environment. Known bounds can be used to simplify the analysis.
- The computed timing constraints are easy to validate. Validation consists on evaluating the linear with the specific constant delay bounds.
- Linear timing constraints provide meaningful information about the system: each constraint is implicitly comparing the delay of two competing paths inside the circuit.

The proposed methodology also has several shortcomings. Due to its roots in abstract interpretation, approximation may appear in the analysis causing false negatives. However, this does not seem to a problem in practical examples. The most relevant drawback is the high complexity of the method, which grows exponentially with the number of symbolic delays in the circuit. The following chapter will address several strategies that reduce the CPU time and memory usage. Nevertheless, exponential complexity continues to be a barrier towards scalability of the method for large circuits, as it happens in all known techniques for the verification of parametric timed systems.

In any case, the complexity of the analysis may be tuned by *balancing* the following factors. First, having known constant delay bounds instead of parameters reduces the genericity and precision of the timing constraints, but improves the computational cost. Also, the level of abstraction where the circuit is studied affects the analysis. The asynchronous controllers presented in this chapter have been studied at the gate-level. However, the technique is also applicable to any level of granularity. For example, one could verify RTL specifications with delays at the level of functional blocks (ALUs, counters, controllers, etc).

Chapter 5

The Octahedron Abstract Domain

Where there is matter, there is geometry.

—Johannes Kepler

This chapter presents a new numerical abstract domain for abstract interpretation called *octahedron*. This abstract domain encodes conjunctions of constraints of the form $(\sum_i c_i \cdot x_i \geq k)$ where $c_i \in \{-1, 0, +1\}$ and $k \in \mathbb{Q}$. These class of constraints, called *unit constraints*, are well suited for the analysis of timed circuits with symbolic delays. Theory, implementation and experimental results are covered in this chapter.

The work presented in this chapter is based on the results published in [50, 51, 53].

5.1 Introduction

In many asynchronous circuits implementing control logic, the timing constraints that arise are unit inequalities. Intuitively, they correspond to constraints of the type

$$\underbrace{(\delta_1 + \cdots + \delta_i)}_{\text{delay(path}_1\text{)}} - \underbrace{(\delta_{i+1} + \cdots + \delta_n)}_{\text{delay(path}_2\text{)}} \geq k$$

indicating that certain paths in the circuit must be slower than other paths. In very rare occasions, coefficients different from ± 1 are necessary. A typical counterexample would be a circuit where one path must be c times longer than another one, e.g. a fast counter.

In the previous chapter, these timing constraints have been analyzed using convex polyhedra. However, as it was discussed in Section 2.4, many different abstract domains can be used to represent the states of a system, each with a different trade-off between precision and efficiency. Several simplifications of convex polyhedra exist, such as *octagons* or the *two variable per inequality* abstract domain. Nevertheless, these domains are based on restricting the number of variables that appear in each constraint, while the timing constraints may be arbitrarily long. Restricting the set of possible coefficients of the variables in a convex polyhedron reduces precision, but it improves the efficiency. Such an abstract domain encodes conjunctions of a finite number of these linear inequalities with unit coefficients ($\{-1, 0, +1\}$), called *unit inequalities* throughout this thesis. The precision of this abstract domain, between octagons and convex polyhedra, motivated the choice of the term *octahedron abstract domain*[1].

Two implementations of this abstract domain are proposed. The main objective behind them is representing a system of unit inequalities without requiring the double description method used for convex polyhedra. Although exponential complexity cannot be avoided, as there is an exponential number of possible unit inequalities, these implementations offer reductions in CPU time and memory usage which allow the analysis of larger examples. An additional assumption which is valid in many problems, the non-negativity of the variables appearing in the unit inequalities, can be used to optimize the implementations even further. The implementations of octahedra can be described as follows:

Decision diagram version: The first implementation attempts to keep a *maximal* representation, encoding all the inequalities implied by the system of constraints. As this set of inequalities may be large, a special type of *decision diagram* called *Octahedron Decision Diagram* (OhDD) is used to reduce the memory usage. As it is usual with decision diagram approaches, the reduction in memory usage is balanced by an increase in execution time. This data structure allows the definition of a fully symbolic version of the verification procedure described in the previous chapter.

Bit-vector version: The second implementation uses the complementary strategy: keeping a *minimal* representation, where all the redundant inequalities have been removed from the system of constraints. Assuming that all variables are non-negative, as it happens in timing analysis and many other verification problems, allows an efficient implementation of the operations using *bit-vectors*. Although the reduction in memory used is not as large as with decision diagrams, the CPU time used by this method is more practical.

The remaining of this chapter will focus on describing the octahedron abstract domain, both implementations, and the different trade-offs between precision and efficiency. Experimental results will illustrate the benefits of both approaches and compare them with the previously described method based on convex polyhedra. We will also discuss other possible applications of the octahedron abstract domain in problems where unit inequalities may arise frequently.

[1]The term *octahedron* is used in geometry to describe a 3-dimensional polyhedron with 8 faces.

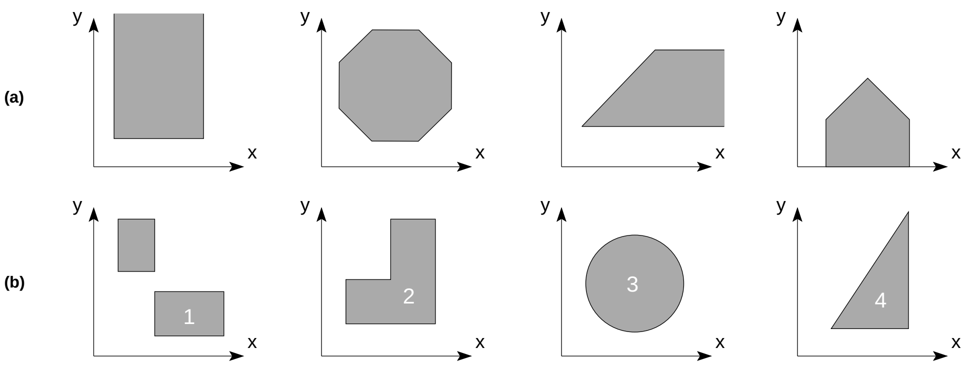


Figure 5.1: Examples of (a) octahedra and (b) non-octahedra over two variables.

5.2 Formal Description of Octahedra

5.2.1 Definitions and Properties

The octahedron abstract domain is now introduced. In the same way as convex polyhedra, an octahedron abstracts a set of vectors in $\mathbb{Q}^n$ as a system of linear inequalities satisfied by all these vectors. The difference between convex polyhedra and octahedra is the family of constraints that are supported.

Definition 5.1 (Unit linear inequality) *A linear inequality is a constraint of the form* $(c_1 \cdot x_1 + \ldots + c_n \cdot x_n \geq k)$ *where the constant term* k *is in* $\mathbb{Q} \cup \{-\infty\}$ *and the coefficients* c_i *are in* $\mathbb{Q}$*, e.g.* $(3x + 2y - z \geq -7)$*. A linear inequality will be called* unit *if all coefficients are in* $\{-1, 0, +1\}$*, such as* $(x + y - z \geq -7)$*.*

Definition 5.2 (Octahedron) *An octahedron* O *over* $\mathbb{Q}^n$ *is the set of solutions to the system of* m *unit inequalities* $O = \{X \mid AX \geq B\}$*, with* $B \in (\mathbb{Q} \cup \{-\infty\})^m$ *and* $A \in \{-1, 0, +1\}^{m \times n}$*. Octahedra satisfy the following properties:*

1. Convexity: *An octahedron is a convex set, i.e. any segment between two points of the octahedron is fully within the octahedron.*
2. Closed for intersection: *The intersection of two octahedra is also an octahedron.*
3. Non-closed for union: *In general, the union of two octahedra might not be an octahedron.*

Figure 5.1(a) shows some examples of octahedra in two-dimensional space. In Fig. 5.1(b) there are several regions of space which are not octahedra, either because they are not connected (1), they are not convex (2), they cannot be represented by a finite system of linear inequalities (3), or because they can be represented as system of linear inequalities, but not unit linear inequalities (4). Notice that in two-dimensional space all octahedra are octagons; octahedra can only show a better precision than octagons in higher-dimensional spaces.

During the remaining of this chapter, we will use C to denote a vector in $\{-1, 0, +1\}^n$ where n is the number of variables. Intuitively, C defines the coefficients of a unit inequality. Therefore, given a set of variables X, the expression $(C^T X \geq k)$ denotes the unit linear inequality $(c_1 \cdot x_1 + \ldots + c_n \cdot x_n \geq k)$.

Lemma 5.1 *An octahedron over n variables can be represented by at most $3^n - 1$ non-redundant inequalities.*

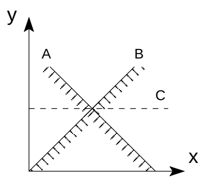
PROOF *Each variable can have at most three different coefficients in a unit linear inequality. For n variables, there at most 3^n possible combinations of unit coefficients, where one of them is an irrelevant unit inequality with 0 in all coefficients. This means that if an octahedron has more than $3^n - 1$ unit inequalities, some of them will only differ in the constant term, e.g. $(C^T X \geq k_1)$ and $(C^T X \geq k_2)$. Only one of these inequalities is non-redundant, the one with the tightest bound (the largest constant), i.e. $(C^T X \geq \max(k_1, k_2))$.* □

A problem when dealing with convex polyhedra and octahedra is the lack of canonicity of the systems of linear inequalities: the same polyhedron/octahedron can be represented with different systems of inequalities. For example, both systems of inequalities $(x = 3) \wedge (y \geq 5)$ and $(x = 3) \wedge (x + y \geq 8)$ define the same octahedron with different inequalities. Given a convex polyhedron, there are algorithms to minimize the number of constraints in a system of inequalities, i.e. removing all constraints that can be derived as linear combinations. However, in the previous example both representations are minimal and even then, they are different. Given that the number of possible linear inequalities in a convex polyhedron is infinite, the definition of a canonical form for convex polyhedra seems a difficult problem. However, a canonical form for octahedra can be defined using the result of lemma 5.1. Even though the number of inequalities of this canonical form makes an explicit representation impractical, symbolic representations based on decision diagrams can manipulate sets of unit inequalities efficiently.

Definition 5.3 (Canonical form of octahedra) *The canonical form of an octahedron $O \subseteq \mathbb{Q}^n$ is either (i) the empty octahedron or (ii) a system of $3^n - 1$ unit linear inequalities, where in each inequality $(C^T X \geq k)$, k is the tightest bound satisfied by O.*

Theorem 5.1 *Two octahedra O_1 and O_2 represent the same subset of $\mathbb{Q}^n$ if and only if they both have the same canonical form.*

PROOF *(Implication $\rightarrow$) Given a constraint $(C^T X \geq k)$, there is a single tightest bound to that constraint. Therefore, if two octahedra are equal, they will have the same bound for each possible linear constraint, and therefore, the same canonical form.* □



$$\begin{aligned}
A &= \{\, (-\infty \le x \le +\infty) \wedge (-\infty \le y \le +\infty) \\
&\quad \wedge (-\infty \le x-y \le +\infty) \wedge (-\infty \le x+y \le 6) \,\} \\
B &= \{\, (-\infty \le x \le +\infty) \wedge (-\infty \le y \le +\infty) \\
&\quad \wedge (0 \le x-y \le +\infty) \wedge (-\infty \le x+y \le +\infty) \,\} \\
A \cap B &= \{\, (-\infty \le x \le +\infty) \wedge (-\infty \le y \le +\infty) \\
&\quad \wedge (0 \le x-y \le +\infty) \wedge (-\infty \le x+y \le 6) \,\} \\
\text{canonical}(A \cap B) &= \{\, (-\infty \le x \le +\infty) \wedge (-\infty \le y \le \mathbf{3}) \\
&\quad \wedge (0 \le x-y \le +\infty) \wedge (-\infty \le x+y \le 6) \,\}
\end{aligned}$$

Figure 5.2: An example where $A \cap B$ is not in canonical form.

PROOF *(Implication ←) From its definition, an octahedron is completely characterized by its system of inequalities. If two octahedra O_1 and O_2 have the same canonical form, then they satisfy exactly the same system of inequalities and therefore are equal.* □

Theorem 5.2 *Let A and B be two non-empty octahedra represented by systems of inequalities of the form $(C^T X \ge k_a)$ and $(C^T X \ge k_b)$ for all $C \in \{-1, 0, +1\}^n$. The intersection $A \cap B$ is defined by the system of inequalities $(C^T X \ge \max(k_a, k_b))$, which might be in non-canonical form even if the input systems were canonical.*

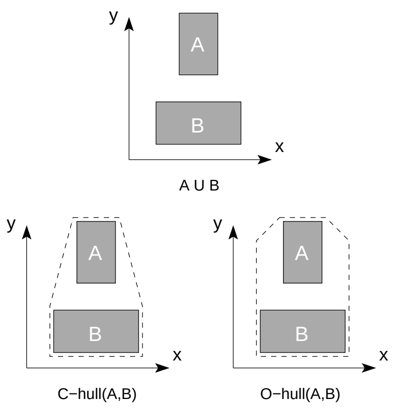
PROOF *Any point $P \in \mathbb{Q}^n$ that satisfies $(C^T P \ge \max(k_a, k_b))$ will also satisfy $(C^T P \ge k_a)$ and $(C^T P \ge k_b)$. Therefore, any point P satisfying the new system of inequalities will also appear in both A and B.* □

Figure 5.2 shows an example where the intersection of two octahedra is not in canonical form, even when the original octahedra were in canonical form. The intersection of $A = \{\, x \ge y \,\}$ and $B = \{\, x + y \le 6 \,\}$ satisfies the inequality $(y \le 3)$. However, this constraint does not appear simply by taking the maximum constant for all the constraints of A and B. Instead, this implicit constraint is implied by the other constraints of the intersection, i.e. it computed as a linear combination of other constraints of $A \cap B$.

Lemma 5.2 *An octahedron B is an upper approximation of an octahedron A, noted $A \subseteq B$, iff (i) A is empty or (ii) for any constraint $(C^T X \ge k_a)$ in the canonical form of A, the equivalent constraint $(C^T X \ge k_b)$ in the canonical form of B has a constant term k_b such that $(k_a \ge k_b)$.*

PROOF *By definition, $A \subseteq B$ iff $A = A \cap B$. This lemma is a direct consequence of this property and Theorem 5.2.* □

Definition 5.4 (Convex and octahedral hull) *The* convex hull (C-hull) *of two convex polyhedra A and B is the intersection of all convex polyhedra that include both A and B. The* octahedral hull (O-hull) *of two octahedra A and B is the intersection of all octahedra that include both A and B.*



$$\begin{aligned}
A &= \{(4 \geq x \geq 2) \wedge (7 \geq y \geq 4)\} \\
B &= \{(5 \geq x \geq 1) \wedge (3 \geq y \geq 1)\} \\
\textsf{C-hull} &= \{(5 \geq x \geq 1) \wedge (7 \geq y \geq 1) \wedge \\
&\quad (4x - y \geq 1) \wedge (-4x - y \geq -23)\} \\
\textsf{O-hull} &= \{(5 \geq x \geq 1) \wedge (7 \geq y \geq 1) \wedge \\
&\quad (x - y \geq 5) \wedge (-x - y \geq -11)\}
\end{aligned}$$

Figure 5.3: Two upper approximations of the union: convex hull (C-hull) and octahedral hull (O-hull)

Figure 5.3 shows an example of the convex and octahedral hulls of two octahedra A and B. Notice that the convex hull is always an upper approximation of the union, and the octahedral hull is always an upper approximation of the convex hull, i.e. $A \cup B \subseteq \textsf{C-hull}(A, B) \subseteq \textsf{O-hull}(A, B)$.

Theorem 5.3 *Let A and B be two non-empty octahedra whose canonical form are respectively $(C^T X \geq k_a)$ and $(C^T X \geq k_b)$ for all $C \in \{-1, 0, +1\}^n$. Then, the octahedral hull O-hull(A, B) is defined by the system of inequalities $(C^T X \geq \min(k_a, k_b))$*

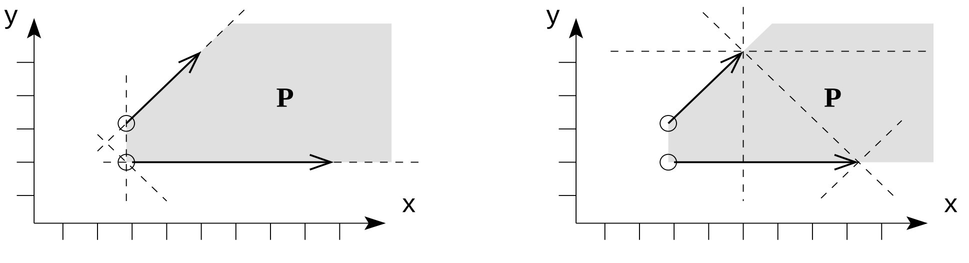
PROOF *Given a bound k for one inequality $(C^T X \geq k)$ of O-hull(A, B), the proof can be split into two parts: proving that $k \leq \min(k_a, k_b)$ and proving that $k \geq \min(k_a, k_b)$.*

As the octahedral hull includes A and B, all points $P \in A$ and $P \in B$ should also be in O-hull(A, B). Therefore, any point in A or B should satisfy the constraints of O-hull(A, B). Given a constraint $(C^T X \geq k)$, it is known that points in A satisfy $(C^T X \geq k_a)$ and points in B satisfy $(C^T X \geq k_b)$. If both sets of points must satisfy the constraint in O-hull(A, B), then k must satisfy $k \leq \min(k_a, k_b)$.

On the other side, the octahedral hull is the least octahedron that includes A and B. Therefore, the bounds of each constraint should be as tight as possible, i.e. as large as possible. If we know that $k \leq \min(k_a, k_b)$ should hold for a given unit inequality, the tightest bound for that inequality is precisely $k = \min(k_a, k_b)$. As a corollary, *the octahedral hull computed in this way is in canonical form.* □

5.2.2 Computing the Canonical Form

The computation of the canonical form of octahedra will be based on the double description method used in convex polyhedra (see Section 2.4.6). More precisely, it is based on the generator representation of the octahedron. The pseudocode for



$$\begin{array}{rcl} x & \geq & 3 \\ y & \geq & 2 \\ x+y & \geq & 5 \\ x-y & \geq & 0 \end{array} \qquad \begin{array}{rcl} x & \leq & \infty \\ y & \leq & \infty \\ x+y & \leq & \infty \\ x-y & \leq & \infty \end{array}$$

(a) **(b)**

Figure 5.4: (a) Bounded unit inequalities, (b) unbounded unit inequalities.

a possible algorithm is presented in Figure 5.5. The output of the algorithm should be either the empty octahedron or the bounds for each of the $3^d - 1$ unit inequalities of the canonical form.

The algorithm is based on the two following observations. First, a ray is a vector that represents a direction of unbounded growth in the octahedron, i.e. no constraint can impose a bound that “crosses” the ray. Therefore, a unit inequality will be unbounded in an octahedron O if it crosses one of the rays of O. And second, if a unit inequality is bounded, its tightest bound will occur in one of the vertices of the octahedron. Figure 5.4 shows an example of these observations in the system of generators of an octahedron with vertices $V = \{(3,3),(3,2)\}$ and rays $R = \{(1,1),(1,0)\}$. All the unit inequalities that cross the rays are unbounded as shown in Fig. 5.4(b). On the other hand, all the bounded inequalities achieve the tightest bound in one of the vertices of the octahedron, as shown in Fig. 5.4(a).

If a unit inequality is defined as $(C^T X \geq k)$, it is possible to determine whether it crosses a ray r using the scalar product: if $(C^T \cdot r < 0)$, then the unit inequality crosses the ray. For example, the ray $(1,0)$ is crossed by the inequality $(-x + y \geq k)$ because $(-1,1)^T \cdot (1,0) = (-1) \cdot 1 + 1 \cdot 0 = -1$. Also, given a vertex v, the tightest bound k achieved by a unit inequality $(C^T X \geq k)$ can be computed as $k = C^T \cdot v$. For instance, the bound of the inequality $(x - y \geq k)$ in a vertex $(3,2)$ is $(1,-1)^T \cdot (3,2) = 1 \cdot 3 + (-1) \cdot 2 = 1$.

For an octahedron over $\mathbb{Q}^d$ with v vertices and r rays, the algorithm requires $O(d \cdot (v + r) \cdot 3^d)$ time. The space requirements are $O(d \cdot (v + r + 3^d))$ in the worst case. In addition to this complexity, the cost of computing the generator representation should be also considered, i.e. $O(c^{\lfloor \frac{d}{2} \rfloor})$, where c is the size of the system of constraints $(c \leq 3^d)$.

Function *Canonical_form*(O)
Input: An octahedron O defined as a system of inequalities over d variables.
Output: The canonical form of O, which is either the empty octahedron (if O is empty) or the canonical system of $3^d - 1$ unit inequalities for O.

```
Compute the system of generators of $O$: the set of vertices $V$ and rays $R$
if $V = \emptyset$ then
  return empty
endif
{ Compute the bound of each inequality in the canonical form }
for each unit inequality $C \in \{-1, 0, +1\}^d$ in $O$ do
  { Evaluate the rays for this inequality }
  unbounded := false
  for each ray $r \in R$ do
    unbounded := $(C^T \cdot r < 0)$?
    if unbounded then
      break
    endif
  endfor
  if unbounded then
    tightest_bound := $-\infty$
  else
    { Evaluate all the vertices for this inequality }
    { Keep the tightest bound satisfied by all vertices }
    tightest_bound := $+\infty$
    for each vertex $v \in V$ do
      new_bound := $C^T \cdot v$
      { $(C^T \cdot X \geq$ new_bound) holds for this vertex }
      tightest_bound := $\min$ ( new_bound, tightest_bound )
      { $(C^T \cdot X \geq$ tightest_bound) holds for all vertices visited so far }
    endfor
  endif
  bounds[ $C$ ] = tightest_bound
  { bounds[ $C$ ] = $k$ means $(C^T \cdot X \geq k)$ }
endfor
return bounds
```

Figure 5.5: Pseudocode to compute the canonical form of an octahedron.

The complexity of these algorithms makes the computation of the canonical form impractical. Even though the canonical form is useful to define the semantics of the operations, it is not practical for the implementation. Instead of the canonical form, we will define a relaxed version called the *saturated form*. Operations performed using the saturated form may lose some precision, while always being an upper approximation of the exact result.

5.2.3 Approximations of the Canonical Form

As it was shown in the previous section, the canonical form of an octahedron provides a useful mechanism to define operations such as the test for inclusion, the intersection or the octahedral hull. However, it is not convenient to implement the operations using the canonical form.

On the other hand, octahedra are defined in the context of abstract interpretation of numerical properties. In this context, the problem is the abstraction of a set of values in $\mathbb{Q}^n$, and the main concern is ensuring that the abstraction is an *upper approximation* of the concrete set of values. Thus, as long as an upper approximation can be guaranteed, an exact representation of octahedra is not required, as octahedra are already abstractions of more complex sets. Keeping this fact in mind, efficient algorithms that operate with upper approximations of the canonical form can be designed.

The first step is the definition of a relaxed version of the canonical form, which is called *saturated* form. While the canonical form has the tightest bound in each of its inequalities, the bounds in the saturated form may be more relaxed. A system of unit inequalities is in saturated form as long as the bounds imposed by the sum of any pair of constraints in the system appear explicitly. For example, a saturated form of the octahedron $(a \geq 3) \wedge (b \geq 0) \wedge (c \geq 0) \wedge (b - c \geq 7) \wedge (a + b \geq 8) \wedge (a + c \geq 6)$ can be defined by the following system of inequalities:

$$\begin{gathered}(a \geq 3) \wedge (b \geq 7) \wedge (c \geq 0) \wedge (a + b \geq 10) \wedge (a + c \geq 6) \wedge (b + c \geq 7) \\ \wedge (b - c \geq 7) \wedge (a + b - c \geq 10) \wedge (a + b + c \geq 13)\end{gathered}$$

where the constraints with a bound of $-\infty$ have been removed for brevity. In this example, saturation has exposed explicitly that $(a + b \geq 10)$. This inequality is the linear combination of $(a \geq 3)$, $(b - c \geq 7)$ and $(c \geq 0)$.

A saturated form O^* of an octahedron $O = \{ X \mid AX \geq B \}$ can be computed using an iterative procedure called *saturation*. At each step of this procedure, a linear combination between two unit inequalities is computed. If this linear combination has a tighter bound than the one already known, the bound is updated, and so on until a fixpoint is reached. The formal description of saturation is the following:

1. Initialize the system of $3^n - 1$ unit inequalities for all possible values of the coefficients $C \in \{-1, 0, +1\}^n$. The bound k of a given inequality

$(C^T X \geq k)$ is chosen as:

$$k = \begin{cases} b & \text{if } C^T X \geq b \text{ appears in } AX \geq B \text{ and } C \not\geq 0^n. \\ -\infty & \text{otherwise} \end{cases}$$

2. Select two inequalities $C_1^T X \geq k_1$ and $C_2^T X \geq k_2$ such that $k_1 > -\infty$ and $k_2 > -\infty$. Let us define $C_* = C_1 + C_2$ and $k_* = k_1 + k_2$.
3. If $C_* \notin \{-1, 0, +1\}^n$ return to step 2.
4. If $C_*^T X \geq k$ appears in the system of inequalities with $k \geq k_*$, return to step 2.
5. Replace the inequality $C_*^T X \geq k$ by $C_*^T X \geq k_*$.
6. Repeat steps 2-5 until:
   - A fixpoint is reached *or*
   - An inequality $C_*^T X \geq k$ with $C = 0^n$ and $k > 0$ is found. In this case, the octahedron is empty.

Theorem 5.4 *Let $O = \{X \mid AX \geq B \ \wedge \ X \geq 0^n\}$ be a non-empty octahedron. The* saturation *algorithm applied to O terminates.*

PROOF *Each step of the saturation algorithm defines a tighter bound for an inequality of the octahedron. The new inequality $(C_3^T X \geq k_3')$ is obtained from two previously known inequalities $(C_1^T X \geq k_1)$ and $(C_2^T X \geq k_2)$, so that $C_3 = C_1 + C_2$ and $k_3' = k_1 + k_2$, and $k_3' > k_3$, where k_3 is the previously known bound for the inequality. If inequalities 1 and 2 were computed in previous rounds of the saturation algorithm, this dependency chain can be expanded, e.g. if inequality 2 comes from inequalities 4 and 5, then $C_3 = C_1 + C_4 + C_5$ and $k_3' = k_1 + k_4 + k_5$. Non-termination of the saturation algorithm implies that there will be infinitely many sums of pairs of inequalities. Ignoring the bound k, there are only finitely many inequalities over n variables. Therefore, it is always possible to find a step that computes a bound k_j' that depends on a previously known bound k_j, i.e. $C_j = C_j + \sum C_l$ and $k_j' = k_j + \sum k_l$. As $C_j - C_j = \sum C_l = 0^n$ and $k_j' - k_j = \sum k_l > 0$, the linear combination $((\sum C_l)^T X \geq (\sum k_l))$ is equivalent to $(0 > 0)$, which implies that O is empty.* □

The fixpoint in saturation *may not be reached* if the octahedron is empty. For example, the octahedron in Fig. 5.6(a) is empty because the sum of the last four inequalities is $(0 \geq 4)$. The saturation algorithm applied to this octahedron does not terminate. Adding the constraints in bottom-down order allows the saturation algorithm to produce $(x_2 - x_4 \geq 5)$, which can again be used to produce $(x_2 - x_4 \geq 9)$ and so on. Even then, the saturation algorithm is used to perform the emptiness test because of three reasons. First, there are special types of octahedra where termination is guaranteed. For instance, if all inequalities describe constraints between

$$\begin{array}{rcl}
+x_2 \qquad -x_4 & \geq & 1 \\
-x_1 - x_2 + x_3 + x_4 + x_5 - x_6 & \geq & 1 \\
+x_1 - x_2 - x_3 + x_4 - x_5 + x_6 & \geq & 1 \\
+x_1 + x_2 + x_3 - x_4 - x_5 - x_6 & \geq & 1 \\
-x_1 + x_2 - x_3 - x_4 + x_5 + x_6 & \geq & 1
\end{array}$$

(a)

$$\begin{array}{rcl}
+x_1 - x_2 - x_3 + x_4 & \geq & 1 \\
-x_1 - x_2 + x_3 + x_5 & \geq & 2 \\
+x_1 + x_2 + x_3 + x_6 & \geq & 3
\end{array}$$

(b)

Figure 5.6: (a) Empty octahedron where the saturation algorithm does not terminate and (b) Non-empty octahedron where the saturated form is different from the canonical form.

symbols (all constant terms are zero), saturation is guaranteed to terminate. This occurs because any linear combination among unit constraints will either leave a constant as 0, or replace a $-\infty$ constant by a 0. Second, the conditions required to build an octahedron for which the saturation algorithm does not terminate are complex and artificial, and therefore we expect them to occur rarely in practical examples. Note that non-termination may arise only for *some* systems of unit inequalities that define an empty octahedron. A final reason is the good behavior of the saturation procedure in practical examples. If the input is an octahedron obtained by performing minor changes to previously saturated octahedra, e.g. the intersection of two saturated octahedra, typically very few iterations are required to reach the fixpoint. This makes the prediction of non-termination possible in practice.

Even if the saturation algorithm terminates, in some cases it might fail to discover the tightest bound for an inequality. For example, in the octahedron in Fig. 5.6(b), saturation will fail to discover the constraint $(x_1 - x_2 + x_3 + x_4 + x_5 + x_6 \geq 6)$, as any sum of two inequalities will yield a non-unit linear inequality. Therefore, given a constraint $(C^T X \geq k_s)$ in the saturated form, the bound k_c for the same inequality in the canonical form may be different, $k_c \not\leq k_s$. But $k_c \geq k_s$ always holds, as k_c is the tightest bound for that inequality. Using this property, operations like the union or intersection that have been defined for the canonical form can also be used for the saturated form. The result will always be an upper approximation of the exact canonical result, as $k_c \geq k_s$ is the exact definition for upper approximation of octahedra (Lemma 5.2).

5.3 Decision-diagram Based Implementation

5.3.1 Overview

The constraints of an octahedron can be represented compactly using a specially devised decision diagram representation. This representation is called *Octahedron Decision Diagram* (OhDD). Intuitively, it can be described as a Multi-Terminal Zero-Suppressed Ternary Decision Diagram:

- *Ternary*: Each non-terminal node represents a variable x_i and has three output arcs, labelled as $\{-1, 0, +1\}$. Each arc represents a coefficient of x_i in a linear constraint.

- *Multi-Terminal* [82]: Terminal nodes can be constants in $\mathbb{R} \cup \{-\infty\}$. The semantics of a path σ from the root to a terminal node k is the linear constraint $(c_1 \cdot x_1 + c_2 \cdot x_2 + \ldots + c_n \cdot x_n \geq k)$, where c_i is the coefficient of the arc taken from the variable x_i in the path σ.

- *Zero-Suppressed* [112]: If a variable does not appear in any linear constraint, it also does not appear in the OhDD. This is achieved by using special reduction rules as it is done in Zero-Suppressed Decision Diagrams.

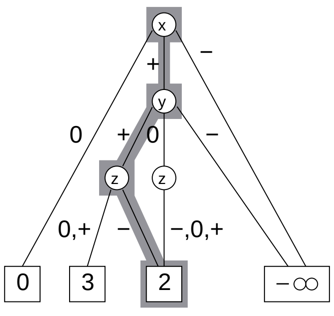
The reduction rules of decision diagrams have an essential role: ensuring that the representation is canonical. In the context of octahedra, canonicity means that saturated octahedra with the same system of inequalities are encoded by the same OhDD. Another aspect of the reduction rules is that they may have a large impact in the efficiency of the representation. A careful choice of reduction rules may decrease the size of the decision diagram, improving both memory and CPU time for all operations. In the case of OhDD, two reduction rules are defined: one for octahedra with unbounded variables and another specially tuned for octahedra with non-negative variables. In both case, the overall manipulation of OhDD will be the same, with only subtle changes in the implementation. These issues will be described in detail in the following section.

Figure 5.7 shows an example of a OhDD and the octahedron it represents on the right, using reduction rules for non-negative variables. The shadowed path highlights one constraint of the octahedron, $(x+y-z \geq 2)$. All constraints that end in a terminal node with $-\infty$ represent constraints with an unknown bound, such as $(x - y \geq -\infty)$. As the OhDD represents the saturated form of the octahedron, some redundant constraints such as $(x + y + z \geq 3)$ appear explicitly.

This representation based on decision diagrams provides three main advantages. First, decision diagrams provide many opportunities for reuse. For example, nodes in a OhDD can be shared. Furthermore, different OhDD can share internal nodes, leading to a greater reduction in the memory usage. Second, the reduction rules avoid representing the zero coefficients of the linear inequalities. Finally, symbolic algorithms on OhDD can deal with sets of inequalities instead of one inequality at a time. All these factors combined improve the efficiency of operations with octahedra.

5.3.2 Notation

Definition 5.5 (Octahedron Decision Diagram - OhDD) *An Octahedron Decision Diagram is a tuple (V, G) where V is a finite set of positive real-valued variables, and $G = (N \cup K, E)$ is a labeled single rooted directed acyclic graph with the following properties. Each node in K, the set of* terminal *nodes, is labeled with a*



$$\begin{aligned} x &\geq 2 \\ y &\geq 0 \\ z &\geq 0 \\ x+y &\geq 3 \\ x-z &\geq 2 \\ x+y-z &\geq 2 \\ x+y+z &\geq 3 \end{aligned}$$

Figure 5.7: An example of a OhDD. On the right, the constraints of the octahedron.

constant in $\mathbb{Q} \cup \{-\infty\}$*, and has an out-degree of zero. Each node* $n \in N$ *is labeled with a variable* $v(n) \in V$*, and it has three outgoing arcs, labeled* $-$*,* 0 *and* $+$*.*

By establishing an order among the variables of the OhDD, the notion of *ordered* OhDD can be defined. The intuitive meaning of ordered is the same as in BDDs, that is, in every path from the root to the terminal nodes, the variables of the decision diagram always appear in the same order. For example, the OhDD in Fig. 5.7 is an ordered OhDD.

Definition 5.6 (Ordered OhDD) *Let* $\succ$ *be a total order on the variables* V *of a OhDD. The OhDD is ordered if, for any node* $n \in N$*, all of its descendants* $d \in N$ *satisfy* $v(d) \succ v(n)$*.*

In the same way, the notion of a *reduced* OhDD can be introduced. However, the reduction rules will be different in order to take advantage of the structure of the constraints. In an octahedron, most variables will not appear in all the constraints. Avoiding the representation of these variables with a zero coefficient would improve the efficiency of OhDD. This can be achieved as in ZDDs by using a special reduction rule.

Let us consider an octahedron like $(x - y \geq 4)$. Other variables, e.g. z, do not affect the bound of the constraint. For example, as there is no information about z, constraints that involve x, y and z will have a bound like $-\infty$, like $(x - y + z \geq -\infty)$ or $(x - y - z \geq -\infty)$. This scenario can be described as follows: there is a node n in the OhDD with a variable z, where the outgoing arcs $-$ and $+$ point towards $-\infty$, and the 0 arc points to a node m. In this case, the node n can be replaced by node m to avoid encoding the irrelevant variable z. This reduction rule is displayed in Figure 5.8(a).

If the variables are known to be non-negative, the reduction rule can be refined. For example, in the case of a constraint like $(x - y \geq 4)$ and an irrelevant variable z, the constraints where the variable z appears are $(x - y + z \geq 4)$ and $(x - y - z \geq -\infty)$. Contrary to the previous case with arbitrary variables, the constraint $(x - y + z \geq 4)$ has now a known bound as $(z \geq 0)$. Therefore, the reduction

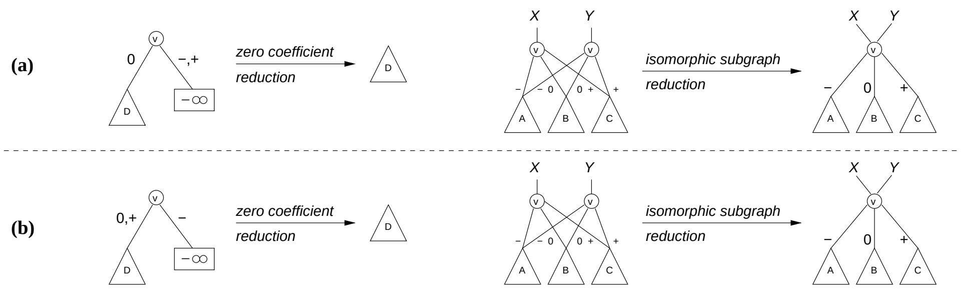


Figure 5.8: Reduction rules for OhDD: (a) Unconstrained variables, (b) non-negative variables

rule should be rephrased to take into account this information: there should be a node n in the OhDD with a variable z, where the outgoing arc $-$ points to ∞ and both the 0 and $+$ arcs point to a node m. Then, the node n can be replaced by m. Figure 5.8(b) depicts this alternative reduction rule. Remarkably, using this reduction rule, the set of constraints stating that "any sum of variables is greater or equal to zero" is represented only as the terminal node 0.

Figure 5.8 shows an example of the two alternatives, together with the other reduction rule, which merges isomorphic subgraphs of the decision diagram. Notice that contrary to BDDs, nodes where all arcs have the same target will not be reduced.

Definition 5.7 (Reduced OhDD) *A reduced OhDD is an ordered OhDD where none of the following rules can be applied:*

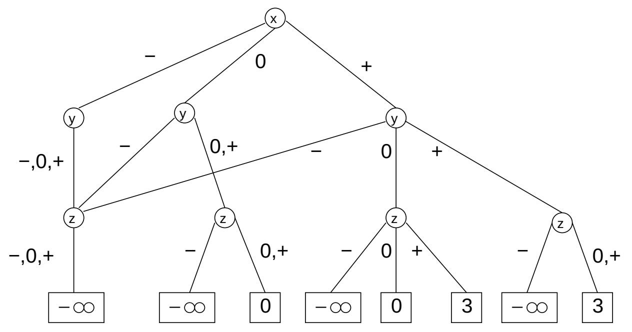
1. Reduction of isomorphic subgraphs*: Let D_1 and D_2 be two isomorphic subgraphs of the OhDD. Merge D_1 and D_2.*

2. Reduction of zero coefficients (with unconstrained variables)*: Let $n \in N$ be a node with the $-$ and $+$ arcs going to the terminal $-\infty$, and with the arc 0 pointing to a node m. Replace n by m* or

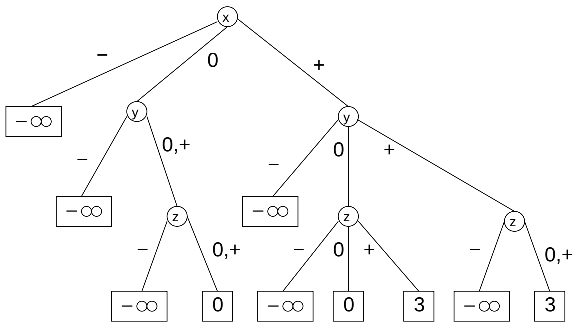
3. Reduction of zero coefficients (with non-negative variables)*: Let $n \in N$ be a node with the $-$ arc going to the terminal $-\infty$, and with the arcs 0 and $+$ point to a node m. Replace n by m.*

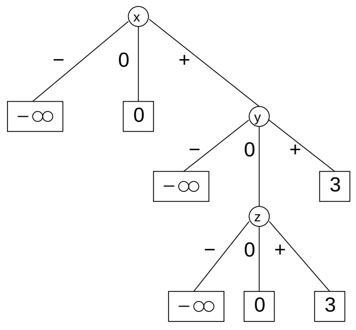
Figure 5.9 shows the effect of the different reduction rules on a OhDD. In this case, the octahedron being represented is $(x \geq 0) \wedge (y \geq 0) \wedge (z \geq 0) \wedge (x + y \geq 3) \wedge (x + z \geq 3)$. The detection of isomorphic subgraphs is illustrated in Figure 5.9(a). As all the variables in this example are non-negative, the two alternative reduction rules 2 and 3 can be compared, as shown in Figures 5.9(b) and (c). Notice that the reduction that assumes non-negative variables produces a more compact representation. Therefore, whenever all the variables in a problem are known to be non-negative, reduction rule number 3 should be used instead of reduction rule number 2.

$$(x \geq 0) \wedge (y \geq 0) \wedge (z \geq 0) \wedge (x + y \geq 3) \wedge (x + z \geq 3)$$



(a)





(b) **(c)**

Figure 5.9: Comparison of reduction rules. On the top, the octahedron being encoded. On the bottom, (a) OhDD with reduction rule 1 - isomorphic subgraphs, (b) OhDD with reduction rule 2 - unconstrained variables and (c) OhDD with reduction rule 3 - non-negative variables.

5.3.3 Related Work

Several classes of decision diagrams have been proposed to encode and manipulate numerical properties. For instance, the decision diagrams presented in Section 3.2.3 can be used to analyze timed systems. However, only constraints with at most two variables can be encoded in these diagrams. The diagrams presented in Section 3.2.5 for the analysis of parametric timed systems can encode an arbitrary number of variables per constraint. This section will recall their characteristics and compare them to OhDD.

Example 5.1 *Figure 5.10 compares the three types of decision diagrams for a given octahedron. Notice that in both DDC and HRD, the nodes of the decision diagram are inequalities of the octahedron. The encoding used in OhDD is radically*

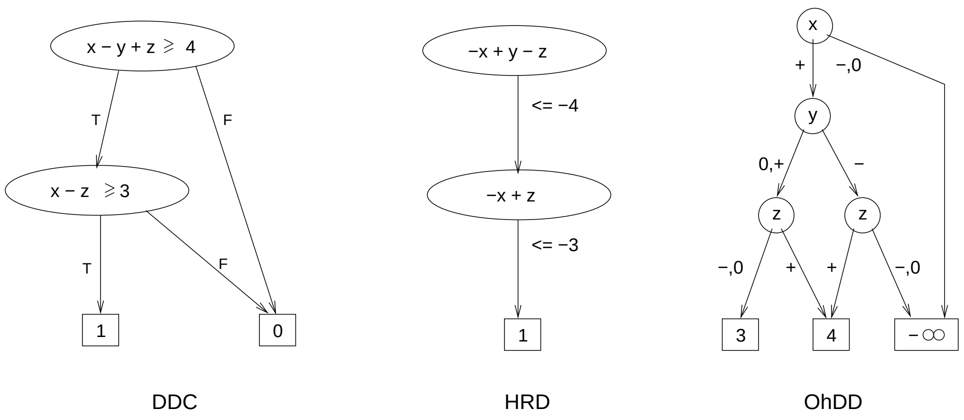


Figure 5.10: Comparison of DDC, HRD and OhDD encoding the octahedron with inequalities $(x - y + z \geq 4) \;\wedge\; (x - z \geq 3)$. The constraints $(x \geq 0)$, $(y \geq 0)$ and $(z \geq 0)$ are not encoded in the DDC and HRD for brevity.

different: each node encodes a variable of the decision diagram, while the output edges model the possible coefficients of the variable. Another important difference is that ability of OhDD to captures many implicit inequalities in the diagram. For instance, the inequality $(x \geq 3)$ *is stored in the OhDD.*

DDC and HRD have two advantages with respect to OhDD: they can encode linear inequalities in addition to unit inequalities, and they can encode non-convex regions, while OhDD can only store octahedra, which are convex by definition. On the other hand, OhDD have a *bounded depth*: each path from the top to the bottom of the OhDD will traverse at most one node per variable in the system. In contrast, the depth of a DDC and HRD may grow with the number of inequalities in a system, e.g. consider an octahedron defined as the intersection of 100 inequalities. Furthermore, OhDD has a systematic procedure to combine the inequalities stored in the diagram, in order to discover all the implicit information: saturation. Meanwhile, DDC and HRD employ heuristics to combine information in inequalities which are close within the diagram. Moreover, in OhDD the order of the nodes in the diagram has a clearer definition than in DDC or HRD. A very simple heuristic can be used to define a good ordering in the OhDD: *"keep the variables that appear in many constraints close to the top of the diagram"*. Such a clear ordering is difficult to establish in diagrams like DDC and HRD.

5.3.4 Abstract Semantics of the Operations

In order to characterize the octahedron abstract domain, the abstract semantics of the abstract interpretation operators must be defined. Intuitively, this abstract semantics is defined as simple manipulations of the saturated form of octahedra. All operations are guaranteed to produce upper approximations of the exact result, as it was justified in section 5.2.3. Some operations like the intersection can deal with non-saturated forms without any loss of precision, while others like the union can only do so at the cost of additional over-approximation.

In the definition of the semantics, A and B will denote octahedra, whose saturated forms contain inequalities of the form $(C^T X \geq k_a)$ and $(C^T X \geq k_b)$, respectively.

- **Intersection** $A \cap B$ is represented by a system of inequalities where $(C^T X \geq \max(k_a, k_b))$, which might be in non-saturated form.
- **Union** $A \cup B$ is approximated by the saturated form $(C^T X \geq \min(k_a, k_b))$.
- **Inclusion** Let A and B be two octahedra. If $k_a \geq k_b$ for all inequalities in their saturated form, then $A \subseteq B$. Notice that the implication does not work in the other direction, i.e. if $k_a \ngeq k_b$ then we don't know whether $A \subseteq B$ or $A \nsubseteq B$.
- **Widening** $A \nabla B$ is defined as the octahedron with inequalities $(C^T X \geq k)$ such that k:

$$k = \begin{cases} -\infty & \text{if } \ k_a > k_b \\ k_a & \text{otherwise} \end{cases}$$

  As established in [114], the result should *not* be saturated in order to guarantee convergence in a finite number of steps.
- **Extension** An octahedron O can be extended with a new variable y by modifying the constraints of its saturated form O^*. Let $(c_1 \cdot x_1 + \ldots + c_n \cdot x_n \geq k)$ be a constraint of O^*, the inequalities that will appear in the saturated form of the extension are:
  - $c_1 \cdot x_1 + \ldots + c_n \cdot x_n - 1 \cdot y \geq -\infty$
  - $c_1 \cdot x_1 + \ldots + c_n \cdot x_n + 0 \cdot y \geq k$
  - $c_1 \cdot x_1 + \ldots + c_n \cdot x_n + 1 \cdot y \geq -\infty$

  If the new variable is known to be non-negative, the last constraint can be changed to a more precise $(c_1 \cdot x_1 + \ldots + c_n \cdot x_n + 1 \cdot y \geq k)$ as the known bound k cannot be decreased by adding a non-negative value.
- **Projection** A projection of an octahedron O removing a dimension x_i can be performed by removing from its saturated form O^* all inequalities where x_i has a coefficient that is not zero.
- **Unit linear assignment** A unit linear assignment $[x_i := \sum_{j=1}^{m} c_j \cdot x_j]$ with coefficients $c_i \in \{-1, 0, +1\}$ can be defined using the following steps:
  - Extend the octahedron with a new variable t.
  - Intersect the octahedron with the octahedron $(t = \sum_{j=1}^{m} c_j \cdot x_j)$

- Project the variable x_i.
- Rename t as x_i.

Impact of the conservative inclusion test on abstract interpretation: Using these operations, upper approximations of the concrete values will be computed in abstract interpretation. A special mention is the case of test of inclusion, where the result is only definite if the answer is true. Intuitively, this lack of accuracy appears from the impossibility to discover the tightest bound with saturation. In abstract interpretation, the analysis is performed until a fixpoint is reached, and the fixpoint is detected using the test for inclusion. The inaccurate test of inclusion might lead to additional iterations in the abstract interpretation loop. Each iteration will add new constraints to our octahedra that were not being discovered by saturation, until the test for inclusion is able to detect the fixpoint. However, in practical examples, this theoretical scenario does not seem to arise, as constraints tend to be generated in a structured way that allows saturation to obtain good approximations of the exact canonical form.

5.3.5 Implementation in OhDD

The octahedra abstract domain and its operations have been implemented as OhDD on top of the CUDD decision diagram package [146]. Each operation on octahedra performs simple manipulations such as computing the maximum or the minimum between two systems of inequalities, where each inequality is encoded as a path in a OhDD.

Two concepts from BDDs are used to present the implementation of the operations. First, the *top variable* of a OhDD is the variable that appears in the root of the OhDD. Two OhDD may have different top variables, because some variables are not encoded when the reduction rules are applied. Given several ordered OhDD, the top variable among all of them is the one which appears before in the ordering. The other concept is the term *cofactor* from Boolean algebra. The two cofactors of a Boolean formula $f(x_1, \ldots, x_n) : \mathbb{B}^n \rightarrow \mathbb{B}$ are the pair of formulas obtained by replacing variable x_i by constants 0 and 1 respectively inside f. In a BDD, the cofactors of a node f with respect to the top variable are the two children of f. In the context of OhDD, the term cofactor is also used to denote the children of a node. Each node f of the decision diagram has three cofactors f^-, f^0 and f^+ with respect to a variable x. Each cofactor denotes the set of inequalities in f where the variable x has a given coefficient, i.e. f^0 contains all the inequalities where x does not appear. The cofactors of a OhDD f with respect to the top variable are the targets of the three arcs $-$, 0 and $+$, while the cofactors for the other variables are defined by the chosen reduction rule.

The operations on octahedra can be implemented as recursive procedures on a OhDD. The algorithm may take as arguments one or more decision diagrams, depending of the operation. Appendix A presents some algorithms that perform operations on OhDD, such as the reduction rules, intersection and saturation. For the sake of brevity, in this section we will simplify discuss the overall structure of these procedures, which is common to all of them:

1. Check if the call is a base case, e.g. all arguments are constant decision diagrams. In that case, the result can be computed directly.

2. Look up the cache to see if the result of this call was computed previously and is available. In that case, return the precomputed result.

3. Select the top variable t in all the arguments according to the ordering. The algorithm will only consider this variable during this call, leaving the rest of the variables to be handled by the subsequent recursive calls.

4. Obtain the cofactors of t in each of the arguments of the call.

5. Perform recursive calls on the cofactors of t.

6. Combine the results of the different calls into the new top node for variable t.

7. Store the result of this recursive call in the cache. Future calls to this method with the same arguments will use the cached result instead of repeating the computation.

8. Return the result to the caller.

The saturation algorithm is a special case: all sums of pairs of constraints are computed by a single traversal; but if new inequalities have been discovered, the traversal must be repeated. The process continues until a fixpoint is reached. Even though this fixpoint might not be reached, as seen in Fig. 5.6, the number of iterations required to saturate an octahedron tends to be very low (1-4 iterations) if it is derived from saturated octahedra, e.g. the intersection of two saturated octahedra.

These traversals might have to visit 3^n inequalities/paths in the OhDD in the worst case. However, as OhDD are directed graphs, many paths share nodes so many recursive calls will have been computed previously, and the results will be reused without the need to recompute. The efficiency of the operations on decision diagrams depends upon on two very important factors. The first one is the *order of the variables* in the decision diagram. Intuitively, each call should perform as much work as possible. Therefore, the variables that appear early in the decision diagram should discriminate the result as much as possible. A second factor in the performance of these algorithms is the *effectivity of the cache* to reuse previously computed results.

5.4 Bit-vector Based Implementation

5.4.1 Overview

The decision diagram implementation of octahedra manipulates the system of unit inequalities considering also all *implicit* inequalities. The large number of potential inequalities requires an efficient data structure to manipulate them effectively. Another approach consists in considering a minimal system of constraints where all redundant inequalities are removed. In convex polyhedra, an exact method to minimize the system of constraints is described, based on the double description method. A problem of this method is that one representation may have a size which is exponential in terms of the other. In contrast, the implementation of octahedra presented in this section relies only on the system of constraints to perform all the operations. This difference produces a loss of precision but improves the efficiency of the method.

Each constraint is implemented with a pair of *bit-vectors*: one that stores the set of variables with a coefficient $+1$ and another for the variables with a coefficient -1. Assuming that the variables involved in the inequality are non-negative, there is an efficient implementation for many operations over unit constraints using only bit-wise set operations.

Let us consider an architecture with B bits per word. Typical values of B are 16, 32 or 64 in the current technology. An inequality will use $2\lceil \frac{n}{B} \rceil$ words plus the size of the constant. In contrast, linear inequalities are represented as a vector of integers, with one integer per variable and one for the constant term. This requires $(n+1)$ integers for each constraint, much more than the memory used by bit-vectors.

Also, bit-vectors are more efficient in terms of time. All the operations on unit constraints require only $2\lceil \frac{n}{B} \rceil$ bit-wise operations. On the other side, the manipulations of linear constraints require at least n integer operations, potentially including integer sums, products, divisions and computations of the greatest common divisor.

Furthermore, some convex polyhedra packages represent integers with arbitrary precision [17, 99]. Operating with arbitrary precision integers may require more than one CPU cycle and use more memory, so in terms of efficiency the comparison is even more favorable to the bit-vector implementation.

Instead of the vector definition of unit inequalities that were useful to describe the canonical form and the semantics for the OhDD implementation, the bit-vector implementation will use a set-based definition. The following sections will introduce this notation and all the theory required to describe the bit-vector implementation, together with a description of the abstract semantics of the operations. Special focus will be devoted to the potential sources of loss of precision.

5.4.2 Notation

Definition 5.8 (Unit inequality) *A* unit inequality *over a set of variables X is a constraint of the form*

$$\sum_{x \in P} x - \sum_{y \in N} y \geq k$$

where P and N are sets of variables ($P \subseteq X, N \subseteq X$) and k is the constant term ($k \in \mathbb{Q}$). Any unit inequality can be characterized by the triple $\langle P, N, k \rangle$.

Example 5.2 *The constraint $(x + z - y \geq 2)$ is a unit inequality than can be characterized as the triple $\langle \{x, z\}, \{y\}, 2 \rangle$. Only well-formed unit inequalities where the sets P and N are disjoint will be considered. For instance, $(a + b - b - d \geq 3)$ can be rewritten into the equivalent $(a - d \geq 3)$.*

Many of the following definitions will only consider unit inequalities over non-negative values $(\forall x_i \in X : x_i \geq 0)$. This restriction can be imposed in the problem of the analysis of timed systems, and several other analysis problems. Using this restriction will allow convenient definitions and an efficient implementation of the underlying operations.

Definition 5.9 (Implication) *A unit inequality $A = \langle P_A, N_A, k_A \rangle$* implies *a unit inequality $B = \langle P_B, N_B, k_B \rangle$, noted $A \rightarrow B$, if B is true whenever A is true. If both inequalities are defined over non-negative variables, then A implies B if and only if $P_A \subseteq P_B$, $N_B \subseteq N_A$ and $k_A \geq k_B$.*

Example 5.3 *The inequality $(x - y - z \geq 7)$ implies the inequality $(x + t - y \geq 0)$ because*

$$\begin{aligned} (x - y - z \geq 7) \wedge (z \geq 0) &\rightarrow (x - y \geq 0) \\ (x - y \geq 0) \wedge (t \geq 0) &\rightarrow (x + t - y \geq 0) \end{aligned}$$

However, the inequality $(x + t - y \geq 0)$ does not imply $(y \geq 3)$, for example.

Definition 5.10 (Trivial and infeasible inequalities) *A unit inequality $I = \langle P_I, N_I, k_I \rangle$ over a set of non-negative variables is* trivial *(always true) if and only if $N_I = \emptyset$ and $k \leq 0$. Conversely, it is* infeasible *(always false) if and only if $P_I = \emptyset$ and $k > 0$.*

Example 5.4 *The unit inequality $(-x \geq 2)$ is infeasible because $(x \geq 0)$. On the other side, a unit inequality like $(x + y \geq -1)$ will always be true as $(x \geq 0)$ and $(y \geq 0)$ imply $(x + y \geq 0)$.*

Definition 5.11 (Unit combination) *The* unit combination *of two unit inequalities A and B (noted $A \oplus B$) is the inequality obtained by adding the left-hand sides and the right-hand sides of A and B, e.g.*

$$\begin{array}{rcl} \displaystyle\sum_{x \in P_A} x - \sum_{y \in N_A} y & \geq & k_A \\ \oplus \qquad \displaystyle\sum_{x \in P_B} x - \sum_{y \in N_B} y & \geq & k_B \\ \hline \displaystyle\sum_{x \in P_A} x + \sum_{x \in P_B} x - \sum_{y \in N_A} y - \sum_{y \in N_B} y & \geq & k_A + k_B \end{array}$$

$A \oplus B$ will be a unit inequality iff $(P_A \cap P_B = \emptyset)$ and $(N_A \cap N_B = \emptyset)$. However, if A and B are defined over non-negative values, the restriction $(N_A \cap N_B = \emptyset)$ is not required (see the example below). If $A \oplus B$ is a unit inequality, then it can be characterized as the following triple:

$$\langle (P_A \setminus N_B) \cup (P_B \setminus N_A), (N_A \setminus P_B) \cup (N_B \setminus P_A), k_A + k_B \rangle$$

Example 5.5 *The unit combination is a restricted version of the widely used linear combination of inequalities. For instance, the unit combination of inequalities $(x + w - t \geq 2)$ and $(t - y - z \geq 4)$ is $(x + w - y - z \geq 6)$. In some cases, the unit combination will lead to non-unit inequalities. For example, the unit combination of $(x + y \geq 2)$ and $(x - z \geq 0)$ is the inequality $(2x + y - z \geq 2)$, which is not a unit inequality. When the non-unit coefficient is negative, the non-negativity of the variables can be used to remove the non-unit coefficient. For example, the unit combination of $(x - y \geq 2)$ and $(t + w - y \geq 7)$ is the inequality $(x + t + w - 2y \geq 9)$ which is not unit. However, as $(y \geq 0)$:*

$$\begin{array}{rcl} x + t + w - 2y & \geq & 9 \\ \oplus \qquad\qquad y & \geq & 0 \\ \hline x + t + w - y & \geq & 9 \end{array}$$

a unit inequality can be obtained. Notice that this strategy cannot be used when the non-unit coefficient is positive, as a constraint of the form $(-y \geq 0)$ is not available.

Definition 5.12 (Strongest common constraint) *The* strongest common constraint *of two unit inequalities A and B (noted as $A \sqcup B$) is another inequality C such that:*

- $(A \to C) \ \wedge \ (B \to C)$
- *For any inequality D, $(A \to D \wedge B \to D) \Rightarrow (C \to D)$.*

If the two unit inequalities $A = \langle P_A, N_A, k_A \rangle$ and $B = \langle P_B, N_B, k_B \rangle$ are defined over non-negative variables, then the strongest common constraint C can be defined as $C = \langle P_A \cup P_B, N_A \cap N_B, \min(k_A, k_B) \rangle$.

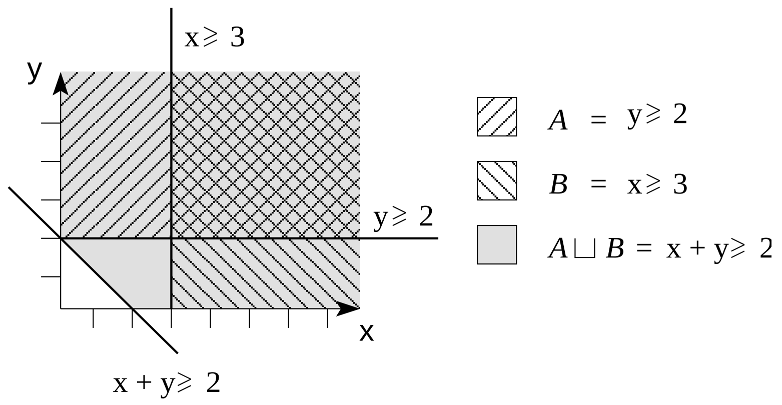


Figure 5.11: A graphical example of the semantics of a strongest common constraint

Example 5.6 *Given* $(x \geq 3)$ *and* $(y \geq 2)$*, the strongest common constraint is* $(x + y \geq 2)$*, as:*

$$\begin{aligned} (x \geq 3) \wedge (y \geq 0) &\rightarrow (x + y \geq 2) \\ (y \geq 2) \wedge (x \geq 0) &\rightarrow (x + y \geq 2) \end{aligned}$$

The values represented by these constraints can be seen graphically in Figure 5.11. Notice that $A \sqcup B$ *does not compute a exact union of the inequalities, but an upper approximation of that union similar to a convex hull. Contrary to a convex hull, the resulting area can be described using only unit inequalities. This notion will be extended in the following section as the* octahedral hull. *Another small example is* $(x + z - y - t \geq 9)$ *and* $(y + z - t \geq 5)$*, whose strongest common constraint is* $(x + y + z - t \geq 5)$*.*

5.4.3 Abstract Semantics of the Operations

An octahedron will be implemented as a finite list of unit inequalities, where each inequality $\langle P, N, k \rangle$ is represented by two bit-vectors (encoding P and N respectively) plus the constant term. All transformations and tests that operate with inequalities will use the set-based definitions from Table 5.1. These definitions allow an efficient implementation using the bit-wise operations of bit-vectors.

The operations required in the timing analysis algorithm are: union ($\cup$), intersection ($\cap$), test for inclusion ($\subseteq$), widening (∇), unit assignment of a variable and existential quantification of a variable. All these operations can be defined in octahedra as transformations of the system of unit inequalities.

Most of these operations require a *satisfiability* test: given a unit inequality $I = \langle P_I, N_I, k_I \rangle$ and an octahedron O, does O satisfy I? A possible implementation of this test is the following:

Table 5.1: Summary of unit inequality operations and tests

$$\sum_{x \in P_A} x - \sum_{y \in N_A} y \geq k_A \qquad \sum_{x \in P_B} x - \sum_{y \in N_B} y \geq k_B$$

$$A = \langle P_A, N_A, k_A \rangle \qquad B = \langle P_B, N_B, k_B \rangle$$

Operation	**Result**
Is A trivial?	true iff $N_A = \emptyset \wedge k_A \leq 0$
Is A infeasible?	true iff $P_A = \emptyset \wedge k_A > 0$
Does A imply B?	true iff $P_A \subseteq P_B \wedge N_B \subseteq N_A \wedge k_A \geq k_B$
Is $A \oplus B$ unit?	true iff $P_A \cap P_B = \emptyset$
$A \sqcup B$	$A \sqcup B = \langle P_A \cup P_B, N_A \cap N_B, \min(k_A, k_B) \rangle$
$A \oplus B$	$A \oplus B = \langle (P_A \setminus N_B) \cup (P_B \setminus N_A),$ $(N_A \setminus P_B) \cup (N_B \setminus P_A), k_A + k_B \rangle$

1. If I is trivial, then O satisfies I.
2. If I is infeasible, then O does not satisfy I.
3. Let N_O be the union of all the sets N from the inequalities in O. If $N_I \not\subseteq N_O$ then O does not satisfy I.
4. If the inequality I is implied by any inequality from O, then O satisfies I.
5. If the inequality I is implied by a unit combination of up to n inequalities from O, then O satisfies I.

The intuitive meaning of step 3 is that a constraint with a variable that does not appear in any inequality of O will not be satisfied by O. For instance, $(x - y - z \geq 4)$ cannot be satisfied by $(x \geq 4) \wedge (t \geq 4)$ as there are no restrictions on y or z. However, this shortcut can only be used for variables appearing with a negative coefficient (N) in the inequality. The non-negativity of variables allows us to add new variables with a positive coefficient. For example, $(x + y \geq 4)$ is satisfied by $(y \geq 8)$ even though the variable x does not appear explicitly. The implicit constraint $(x \geq 0)$ allows us to add x to the inequality.

Step 5 also deserves additional comments. If all the combinations of constraints in O are considered, then the satisfiability test is exact. However, considering all possible unit combinations is too computationally expensive. Instead, a good trade-off between precision and efficiency is achieved when $n = 2$, i.e. only the combinations of pairs of inequalities of O are considered. As a consequence, the satisfiability test will be approximate, while still being *conservative*: some satisfied constraints might be reported as unsatisfied, but not the other way around. In the timing analysis algorithm, this approximation may cause *false negatives* (inability to find sufficient timing constraints, even if they exist) but it will never cause *false positives* (timing constraints will always avoid all errors).

Intersection

The intersection of two octahedra $A = B \cap C$ is defined by the system of unit inequalities with all the inequalities from A and all the inequalities from B. This is the only *exact* operation on octahedra.

Union

The union of two octahedra $A \cup B$ can be approximated as a system of unit inequalities that contains:

- The inequalities from A satisfied by B.
- The inequalities from B satisfied by A.
- The strongest common constraint of all pairs of inequalities from A and B.

Test of inclusion

An octahedra A is included in an octahedron B, noted as $A \subseteq B$, if all the inequalities of B are satisfied by A. Notice that the approximation in the satisfiability test might lead to false negatives in the test of inclusion.

Widening

Widening is the extrapolation operator used to guarantee the termination of the analysis in the presence of loops [63]. The widening of two octahedra $A \nabla B$, where A is the initial property and B is the property after one iteration, extrapolates the result of future iterations based on A and B. The widening $A \nabla B$ for octahedra can be defined as:

- If all the constraints in A and B have a constant term $k = 0$, then $A \cup B$ is a widening operator.
- Otherwise, $A \nabla B$ contains all the inequalities from A that are also satisfied by B.

This definition of widening is also the one used for convex polyhedra in [66].

Unit assignment

Assignments of the form $x' := x + y$ are required in order to perform the timing analysis. After the assignment, we know that $(x \geq y)$ and we also know that the old value of x can be characterized as $x' - y$. Therefore, the assignment should add the constraint $(x \geq y)$ to the system of inequalities of O, and replace each instance of x in the system of inequalities by $x - y$. This replacement is implemented in the following way:

- Inequalities where $x \notin P$ and $x \notin N$ are not modified.
- The unit combinations of all pairs of inequalities $A \oplus B$ of O such that $x \in P_A$ and $x \in N_B$ are added to the system of linear inequalities. This step attempts to minimize the loss of precision: some inequalities might already contain x and y so replacing x by $x - y$ could produce a non-unit inequality. Considering these unit combinations reduces the amount of lost information.
- The inequalities $I = \langle P_I, N_I, k_I \rangle$ where $x \in P_I$ are transformed according to y:
  - If $y \in P_I$, then $P'_I = P_I \setminus \{y\}$.
  - If $y \in N_I$, then I is not modified.
  - Otherwise, $N'_I = N_I \cup \{y\}$.
- The inequalities $I = \langle P_I, N_I, k_I \rangle$ where $x \in N_I$ are transformed according to y:
  - If $y \in P_I$, then I is not modified.
  - If $y \in N_I$, then $N'_I = N_I \setminus \{y\}$.
  - Otherwise, $P'_I = P_I \cup \{y\}$.

After these changes, the constraint $(x \geq y)$ can be added to the system of inequalities.

Existential quantification

The quantification of a variable x will attempt to remove all the known restrictions on x while keeping as much information as possible on the rest of variables. This procedure is implemented using a process called Fourier-Motzkin elimination [67].

Inequalities where $x \notin P$ and $x \notin N$ are unaffected by this procedure. Regarding the remaining inequalities, Fourier-Motzkin proceeds by selecting one constraint where $x \in P$ and one constraint where $x \in N$. The unit combination of these constraints will not contain variable x; if the combination is a unit inequality, it is added to the system of inequalities of O. The final step is the removal of the inequalities of O that do not hold after the quantification. All inequalities where $x \in P$ must be removed, while those with $x \in N$ can be just modified so that $N' = N \setminus \{x\}$. Again, the different behavior of P and N appears from the implicit constraint $(x \geq 0)$ in O.

Table 5.2: Experimental results in the asynchronous pipeline example.

Pipeline example				Polyhedra		OhDD		Bit-vectors	
Stages	$\|P\|$	S	T	CPU	Mem	CPU	Mem	CPU	Mem
2	8	36	88	0s	64Mb	1s	5Mb	0s	1Mb
3	10	108	312	2s	67Mb	17s	8Mb	2s	3Mb
4	12	324	1080	13s	79Mb	249s	39Mb	12s	9Mb
5	14	972	3672	259s	147Mb	1h5min	57Mb	123s	48Mb
6	16	2916	12312	O/M	O/M	39h44min	83Mb	18min	245Mb
7	18	8748	40824	O/M	O/M	T/O	T/O	2h6min	1183Mb

$|P|$ = number of symbolic delays S = number of states T = number of transitions
O/M = out of memory (> 1.5Gb) T/O = timeout (> 48h)

5.5 Experimental Results

5.5.1 Asynchronous Pipeline

The asynchronous pipeline example introduced in Section 4.5.2 will be used again to illustrate the complexity of the approaches presented in this chapter. Table 5.2 compares the results obtained with the bit-vector representation of octahedra, the decision diagram representation (OhDD) and convex polyhedra. Precision is equal for all three approaches as the property to be discovered is formed by unit inequalities. Therefore, the comparison will focus on the execution time and memory usage of the timing verification procedure.

Regarding memory, both OhDD and the bit-vector implementation show improvements with respect to convex polyhedra. The reduction is sufficient to verify longer pipelines, where not only there are more symbols but also the state space is larger. Remarkably, octahedra represented with bit-vectors can be used to analyze pipelines with state spaces one order of magnitude larger than those analyzable with convex polyhedra. The comparison among OhDD and bit-vectors reveals that although OhDD are worse for smaller examples, the memory usage for bit-vectors grows faster as the size of the examples increase. The reason behind this slower growth is that decision diagrams offer many opportunities for reuse, e.g. many nodes within the directed acyclic graphs are shared by several ancestors.

With respect to the CPU time, OhDD exhibit a large increase with respect to convex polyhedra. As it was mentioned in Section 3.2.3, in the context of timed systems decision diagram methods reduce the memory usage at the expense of additional CPU time. The increment in CPU time may make this approach impractical in larger examples. On the other hand, bit-vectors exhibit lower CPU times than convex polyhedra, with a good time vs. memory trade-off.

5.5.2 Asynchronous Controllers

Table 5.3 recalls the characteristics of the asynchronous controllers studied in the previous chapter: the size of the circuit (number of signals and gates), the size of

Table 5.3: Characteristics of the asynchronous controllers

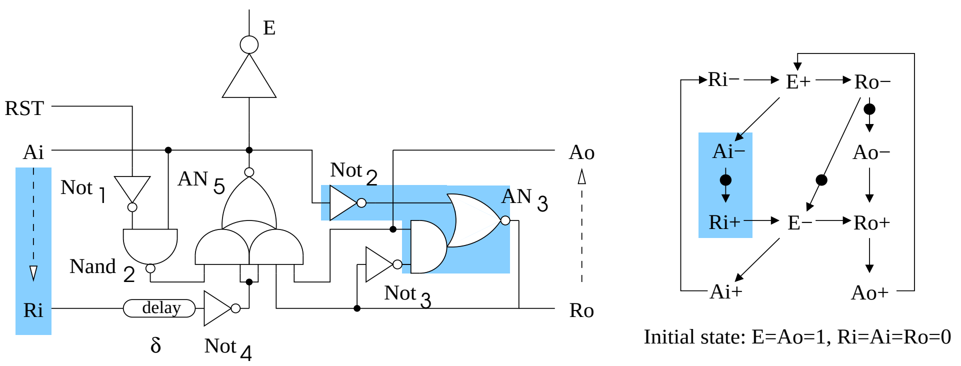
Example	Circuit		STG		PTTS		$\|P\|$
	Wires	Gates	Places	Trans	States	Trans	
nowick	10	7	19	14	60	119	10
gasp-fifo	9	7	10	8	66	209	12
sbuf-read-ctl	13	10	19	16	74	157	14
rcv-setup	9	6	14	15	72	187	12
alloc-outbound	15	11	21	22	82	161	19
ebergen	11	9	16	14	83	188	13
D flip-flop	6	4	16	22	146	448	8
mp-fwd-pkt	13	10	24	16	194	574	12
chu133	12	9	17	14	288	1082	7
desynch	11	8	12	8	304	934	13
converta	14	12	16	14	396	1341	14

Table 5.4: Experimental results for the asynchronous controllers

Example	Convex Polyhedra			
	TC	Sat	CPU	Mem
nowick	2	45	0.5s	83Mb
gasp-fifo	10	28	8.1s	87Mb
sbuf-read-ctl	4	52	1.2s	83Mb
rcv-setup	8	49	2.1s	83Mb
alloc-outbound	3	62	1.3s	83Mb
ebergen	5	61	1,3s	83Mb
D flip-flop	7	112	5.8s	85Mb
mp-fwd-pkt	6	89	1.9s	85Mb
chu133	3	61	1.3s	85Mb
desynch	O/M	O/M	O/M	O/M
converta	13	188	20.4s	92Mb

Example	OhDD				Bit-vectors			
	TC	Sat	CPU	Mem	TC	Sat	CPU	Mem
nowick	=	=	0.1s	8.9Mb	=	=	0.0s	2.6Mb
gasp-fifo	T/O	T/O	T/O	T/O	11	22	4.1s	3.9Mb
sbuf-read-ctl	=	=	1.4s	9.9 Mb	=	=	0.1s	2.9Mb
rcv-setup	=	=	8.3s	21.5Mb	=	=	0.4s	3.0Mb
alloc-outbound	4	61	0.2s	10.2Mb	4	61	0.1s	2.9Mb
ebergen	=	=	1.7s	11.8Mb	=	=	0.1s	2.9Mb
D flip-flop	=	=	68.9s	33.5Mb	=	=	1.6s	4.4Mb
mp-fwd-pkt	=	=	3.8s	20.0Mb	8	82	0.3s	3.8Mb
chu133	=	=	1.0s	12.9Mb	5	56	1.3s	5.5Mb
desynch	6	50	981s	75Mb	6	50	8.0s	4.4Mb
converta	T/O	T/O	T/O	T/O	13	180	138s	15.0Mb

TC = timing constraints Sat = states satisfying the TC Mem = memory usage in Mb
O/M = out of memory (> 1.5Gb) T/O = timeout (> 1h)



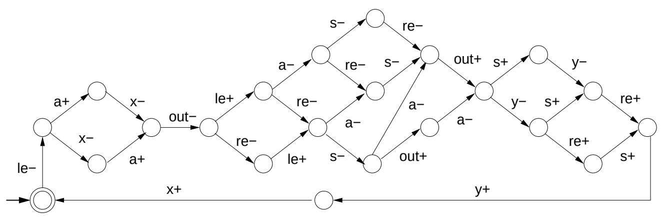
$$\boxed{(Ri+ > Not_2 + AN_3)} \wedge (\delta^+ + Not4 + AN_5 > Not_3)\ \ and$$
$$(Not_3 + Ri- > Not_2 + AN_3 + Ao-)\ \wedge\ (Ao+ > Not_3) \wedge$$
$$(Ao- > Not_3)\ \wedge\ (Not_2 + AN_3 > Nand_2)$$

Figure 5.12: The *desynch* example from Table 5.4, implementing a semi-decoupled controller [27]. The delay δ is asymmetric: δ^+ after $Ri+$, and δ^- after $Ri-$. The highlighted areas correspond to the first timing constraint. The symbolic delay $Ri+$ models the time spent by the environment from the output event $Ai-$ until the input event $Ri+$ occurs. This causality is imposed by the highlighted area of the STG.

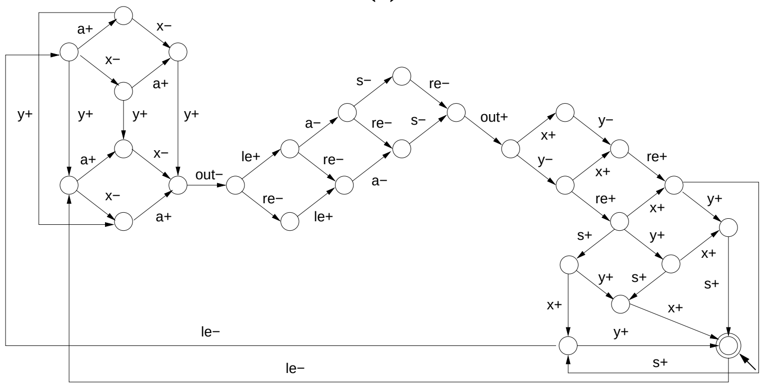
the STG that describes the interaction with the environment, the size of the untimed state space and the number of different symbolic delays ($|P|$) used in the example. A new example, *desynch* is introduced in this table. This circuit is shown in the Figure 5.12.

Table 4.1 shows the experimental results for the verification of these controllers. Convex polyhedra and the two implementations of octahedra are compared using two criteria: precision and efficiency. The first two columns refer to the precision of the timing constraints achieved with each method, while the following two characterize the resources used by the verification algorithm.

In terms of efficiency, the comparison is clearly favorable to octahedra in terms of memory usage. Both bit-vectors and decision diagrams are effective mechanisms to reduce the memory consumption. In terms execution time, the comparison is not so clear. The bit-vector implementation is faster in all case except the last entry *converta*. In this specific circuit, timing constraints with non-unit coefficients are very useful, as some failures are reached when a specific path in the circuit is traversed more than once. Even in this scenario, sufficient unit timing constraints can be found. Moreover, the analysis with convex polyhedra must use additional approximations for this example, as it generates too many constraints and runs out of memory (as it happens in the *desynch* example), while the bit-vector implementation does not have this problem. On the other hand, OhDD sometimes have worse execution time than convex polyhedra. As it is mentioned in the evaluation of the



(a)



(b)

Figure 5.13: Reachable state space of the GasP FIFO controller considering the timing constraints computed with (a) octahedra and (b) convex polyhedra.

asynchronous pipeline example, decision diagrams often trade-off execution time for a reduction in memory. In this case, two circuits could not be verified in less than one hour using OhDD. It should be noted that these circuits are precisely the most complex circuits of the benchmark: *converta* and the GasP FIFO controller.

Quantifying the precision of octahedra versus convex polyhedra is not simple. Obviously, the timing constraints computed by convex polyhedra will be more precise and, therefore, less restrictive. Two indicators have been measured to quantify the difference of precision: the number of timing constraints required for correctness (TC) and the number of states that satisfy these timing constraints (Sat). Intuitively, the second value indicates the degree of restriction imposed by each set of constraints.

Example 5.7 *Let us consider the GasP FIFO controller presented in Figure 4.14. This circuit can be verified using convex polyhedra, but some of the discovered timing constraints are not unit inequalities. Even then, it can also be verified using octahedra. This analysis computes a different set of sufficient timing constraints where all inequalities are unit. However, this set of timing constraints is more restrictive than the one discovered by convex polyhedra. This difference is graphically depicted in Figure 5.13, that shows two subsets of the state space of the GasP FIFO controller. These subsets correspond to the states that satisfy the timing constraints discovered using (a) octahedra and (b) convex polyhedra, i.e. the set of* *acceptable behaviors.*

Notice that some events that may occur concurrently in (b) are forced to occur sequentially in (a) due to the timing constraints. For instance, in (a) $y+$ should occur before $le-$ while in (b) they can occur in any relative order. The difference between both sets of states reflects the quality of the two solution, and therefore, it can be used as an indicator of the precision of both abstract domains. This is the meaning of the Sat column in the Table 5.4.

In several examples, both approaches compute exactly the same constraints (noted as = in the Table). For the other examples, the constraints computed by octahedra are more restrictive. However, the collected data point out that the quality of the constraints computed by both methods is comparable: there are not many additional constraints, nor they are overly restrictive. Note that the precision of OhDD is in some cases better than the precision of bit-vectors, although the difference is not significant. This difference in precision can be explained by the following observation. When performing a union, the bit-vector implementation considers linear combinations of up to 2 inequalities from the system of constraints. Meanwhile, in a OhDD saturation computes *all* the linear combinations of unit inequalities, exposing more implicit information that can be used to improve the precision of the analysis.

5.6 Conclusions

We have presented the octahedron numerical abstract domain, a restricted version of convex polyhedra where all the coefficient of linear inequalities are unitary. This definition provides several benefits, such as a canonical form, which does not exist in convex polyhedra, and efficient implementations. Implementations can be further optimized if the variables in the octahedron are known to be non-negative, as it happens in the analysis of timed systems and many other domains.

The precision of octahedra is between octagons and convex polyhedra. Regarding octagons, octahedra allow constraints with an arbitrary number of variables compared to at most two variables per inequality. With respect to convex polyhedra, there are two sources of loss of precision. First, there is a loss of precision inherent in the restriction of the possible coefficient of linear inequalities. The difference between the octahedral hull and the convex hull exemplify this loss of precision. Furthermore, manipulations outside of the canonical form and avoiding the use of the double description method cause an additional approximation in the results. In spite of this potential loss of precision, octahedra have been sufficient to verify all the examples from the timed circuit domain.

Regarding efficiency, operations on octahedra have a worst-case exponential complexity with respect to the number of variables, like convex polyhedra. In contrast, octagons have a polynomial complexity, due to having at most n^2 inequalities compared to the 3^n inequalities of octahedra. Hence, exponential complexity is inherent to this type of relational numerical abstract domains.

Experimental results have shown the different trade-offs between CPU time and memory in octahedra, and compared them to a convex polyhedron implementation. The decision diagram implementation (OhDD) exhibits a large reduction in memory usage, balanced by an increase in CPU time. Meanwhile, the bit-vector implementation has better CPU time and memory usage than convex polyhedra, while in the largest examples it uses more memory than the OhDD version. The experimental results point to the following conclusion: whenever the precision of octahedra is sufficient (e.g. few non-unit constraints), octahedra should be used instead of convex polyhedra.

Chapter 6

Future Work and Applications

I never think of the future. It comes soon enough.

—Albert Einstein

This chapter presents the future research directions after the work presented in this thesis. Related problems where the verification relies on the analysis of numerical properties will be presented and discussed in connection with the contributions of this thesis.

6.1 Introduction

Previous chapters have shown the application of abstract interpretation to the analysis of a special type of concurrent timed systems: timed circuits. The challenges faced in this verification are common to a larger class of problems.

In many types of systems, a part of the state is described by non-discrete variables, e.g. the values of clocks. Other examples are formalisms that contain some form of counters or variables, like software programs, Petri Nets [119] or automata with counters [12]. As the state space is potentially infinite, abstract interpretation is a suitable approach to approximate the state space. The discovery, representation and manipulation of numerical properties among the state variables is instrumental in this analysis.

This chapter discusses some potential applications of the work described in this thesis, together with possible future extensions and research directions. A case study involving the analysis of Petri Net models is briefly presented, together with preliminary results. This study has been a joint work with Enric Rodríguez-Carbonell, and it has been presented in [54].

6.2 Verification of Timed Circuits

The work described on the verification of timed circuits with symbolic delays has a high computational complexity which limits the size of the circuits that can be studied. Therefore, the main directions of future research focus on improvements in the efficiency and scalability of the approach.

Reductions of the untimed state space: Timing analysis with symbolic delays has a very high complexity. Therefore, simplifying the state space before timing analysis is cost-effective even if we use complex techniques. It is important to guarantee that the constraints discovered by timing analysis are unaffected by these simplifications. Some reductions have been described in Section 4.3 based on the error transitions. Two more concepts can be used to simplify the state space before timing analysis: *nodal points* and *dominators*.

A nodal point is a state of a transition system where all the enabled events were disabled in the predecessor states. As a consequence, all event clocks are reset to zero when the state is reached, meaning that there is no residual information stored in the clocks of a nodal point. Meanwhile, the concept of dominators and post-dominators [88, 107] is widely used in compiler theory for the analysis of control-flow graphs. A node x of a directed graph dominates a node y if any path from the initial node to y must go through x. In a similar way, a definition of post-dominator can be proposed. These concepts provide useful information about the structure of the state space that can be used in the simplification.

For instance, we can perform a fusion of sequential events if there are no other events enabled during the sequence. This transformation does not affect the timing constraints discovered by the verification algorithm, as constraints arise when there is a choice among several enabled events. Furthermore, concurrency diamonds of the form *"either x and then y, or y and then x"* can be reduced if there are no enabled events in the intermediate states. Again, the reduction does not alter the timing constraints as the error is reached regardless of whether x is fired before y or vice versa. Nodal points and dominator information can be used to detect these patterns and generalize the scenarios where reductions can be applied.

Automatic abstraction: The proposed method for the verification of timed circuits works at the gate level. However, it is also applicable for a verification at a higher level of abstraction, e.g. each event corresponds to a complex gate. It would be desirable to have an automatic procedure that automates the abstraction process, identifying blocks of combinational logic that are suitable for being abstracted.

Compositional verification: A useful strategy to scale complex verification techniques is *compositional verification*. When a system is too large to be verified, it is partitioned into smaller elements. Several methods can be used to reason about the correctness of the global system using only local information about the subsystems and their composition. In this context, methods like the *assume-guarantee* paradigm [55] will be explored to improve the scalability of the proposed verification algorithm.

6.3 The Octahedron Abstract Domain

6.3.1 Future Work

Several open problems regarding the octahedron abstract domain are the following:

Efficient algorithms to compute the canonical form: The proposed algorithm for octahedra is based on the double-description method, like convex polyhedra. However, due to its high complexity, it is not practical for being used in the implementation of octahedra operations. Finding efficient algorithms for the computation of the canonical form would improve the precision of the operations, and potentially the efficiency as well. Stronger versions of the saturation procedure provide a starting point for this research direction.

Exact algorithms for octahedral operations: Several operations on octahedra exhibit a loss in precision which is not caused by the shift from linear to unit inequalities. For example, in the bit-vector implementation, the test that checks whether a unit inequality is satisfied by an octahedron is an approximate operation. An exact version of this test would lead to an increase in precision in the operations that use it, even though it is unclear whether the efficiency of such a test would make the implementation practical. A possible approach that we are considering to implement this test is transforming it into an instance of the Boolean satisfiability problem, for which very efficient techniques are available [116].

Study of the double-description of octahedra: In this direction, our goal will be establishing tighter upper bounds for the possible number of vertices and lines in octahedra. The best known upper bound on the number of vertices of a convex polyhedron is $O(c^{\lfloor \frac{n}{2} \rfloor})$, where c is the number of constraints and n is the number of dimensions. In the context of octahedra, this provides a worst-case bound of $O(3^{n^{\lfloor \frac{n}{2} \rfloor}})$. However, given that the only unit inequalities are allowed in octahedra, it is likely that a tighter bound may be defined, even something closer to 3^n. Knowledge of this bound may lead to alternative algorithms to implement octahedra operations.

6.3.2 Potential Areas of Application

The examples presented so far in this thesis have studied the application of octahedra to the verification of parametric timed systems. However, the octahedron abstract domain has also applications in other problems. In general, the octahedron abstract domain may be interesting in any analysis problem where convex polyhedra can be used. Many times, the precision obtained with convex polyhedra is very good, but the efficiency of the analysis limits the applicability. In these scenarios, using octahedra might be adequate as long as unit linear inequalities provide sufficient information for the specific problem. It may also be interesting to ensure that the variables involved in the analysis are non-negative in order to take advantage of the available improvements for the non-negative case. Some examples of areas of applications are the following:

- *Static discovery of bounds in the size of asynchronous communication channels*: Many systems communicate using a non-blocking semantics, where the sender does not wait until the receiver is ready to read the message. In these systems, each channel requires a buffer to store the pending messages. Allocating these buffers statically would improve performance but it is not possible, as the amount of pending messages during execution is not known in advance. Analysis with octahedra could discover these bounds statically. The analysis of the bounds of these channels can be performed using octahedra, as the size of channels is positive. This problem is related to the problem of structural boundedness of a Petri Net [118], where an upper bound on the number of tokens that can be in each place of the Petri Net must be found.

- *Performance analysis of timed systems*: Clocks and delays are restricted to positive values in many types of models. Octahedra can be used to analyze these values and discover complex properties such as timing constraints or worst-case execution time (WCET).

- *Analysis of string length in programs* [74]: Checking the absence of buffer overflows is important in many scenarios, specially in the applications where security is critical, e.g. an operating system. C programs are prone to errors related to the manipulation of strings. Several useful constraints on the length of strings can be represented with octahedra. For instance, a constraint on the concatenation of two strings can be the following:

$$\texttt{strlen}(\texttt{strcat}(s_1, s_2)) = \texttt{strlen}(s_1) + \texttt{strlen}(s_2)$$

- *Analysis of term size in logic programs* [151], which can be used among other things to prove termination of logic programs [145].

- *Proof of mutual exclusion and other synchronization properties among concurrent processes*: many high-level synchronization constraints can be expressed easily as properties on counter (semaphore) variables [83]. For instance, mutual exclusion among n processes can be represented with constraints like $(x_1 + \ldots + x_n \leq 1)$, where $x_i = 1$ if the process i is inside the critical section, and $x_i = 0$ otherwise.

6.4 Case Study: Analysis of Petri Net Models

6.4.1 Motivation

Petri Nets [119] are a widely used formalism for the specification of concurrent systems. In Section 4.2.3, Petri Nets were used to model the interaction between an asynchronous circuit and the environment. However, there are many other areas of application of this formalism, as it able to specify interesting properties like sequential composition, parallel composition, non-deterministic choice or mutual exclusion among others. Moreover, there are formal methods available to study many interesting properties about Petri Nets, e.g. deadlock freedom.

Let us briefly recall the basic notions presented previously in Section 4.2.3. A Petri Net can be briefly described as a directed bipartite graph. The nodes of the graph are of two classes, *places* or *transitions*, and the edges among them have an integer label called *weight*. Each place can contain a number of *tokens*. The state of a Petri Net, called *marking*, is an assignment of a number of tokens to each place. The initial count of tokens is called the *initial marking*. From that marking, the state can evolve by *firing* transitions. Firing a transition t consumes tokens from the places with edges that end in t, while it also produces tokens in the places with an edge coming from t. The number of tokens being produced or consumed is specified by the weight of the edges. A transition can only be fired if it is *enabled*, i.e. if there are enough tokens to be consumed in the input places of the transition. Firing a transition may enable or disable other transitions.

The *reachability problem* pursues the characterization of the set of markings that can be generated by firing enabled transitions from the initial marking. Petri Nets suffer from the state explosion problem, thus the size of the state space can be very large even for moderately-sized specifications. Furthermore, the number of tokens in each place is not bounded in general, so potentially the set of reachable markings may be infinite. There are special classes of nets, called *k-bounded*, where the number of tokens in each place is less or equal to k. However, we will focus on potentially unbounded nets, as the verification of bounded Petri Nets is already addressed by implicit methods based on decision diagrams, e.g. [132].

Several techniques can be used to address the reachability problem in addition to an strict enumeration. For instance, *structural analysis* reveals properties about the underlying Petri Net structure which are independent of the initial marking [143]. Other approaches avoid computing the state space and, instead, discover *invariants* satisfied in all the reachable markings. These invariants may be sufficient to prove several properties about the system, e.g. boundedness of some places, mutual exclusion or deadlock freedom. Some techniques used in this context are the exact analysis using Presburger arithmetics [81] or real arithmetic [24], and inductive invariants generated using Farka's lemma [140]. Abstract interpretation also seems a suitable technique to study this problem, provided that an adequate numerical abstract domain can be defined.

6.4.2 Choice of an Abstract Domain

The characteristics of a Petri Net have an impact on the selection of the best abstract domain for the analysis. The following is a list of observations relevant to the choice:

- The number of tokens in a place is a *non-negative integer* value. In general, this value may be unbounded.
- The number of tokens in a place can only be incremented by a constant, decremented by a constant or compared with a constant value. No more types of abstract assignments and abstract test functions are required.
- Among others, the type of invariants that are discovered during the analysis describe the following properties:
  - *Traps*: The number of tokens in a set of places does not decrease from the initial marking, e.g. $(x_1 + \ldots + x_n \geq k)$.
  - *Siphons*: The number of tokens in a set of places does not increase from the initial marking, e.g. $(x_1 + \ldots + x_n \leq k)$.
  - *Boundedness*: A place or set of places contains a number of tokens within a constant lower and upper bound, e.g. $(k_1 \leq x \leq k_2)$.
  - *Circuits*: The number of tokens in a set of places remains constant in any marking, e.g. $(x_1 + \ldots + x_n = k)$.
  - *Disjunctions*: Several properties can only proved by invariants that capture a choice within the Petri Net, e.g.

    $$(x_1 = 2 \wedge x_2 = 3) \ \vee \ (x_1 = 3 \wedge x_2 = 2)$$

  - *Mutual exclusion*: This is a special case of disjunction of properties. A set of places P contains tokens if and only if another set of places Q does not contain any token, e.g. when $P = \{x_1\}$ and $Q = \{x_2\}$:

    $$(x_1 = 0 \wedge x_2 = 1) \ \vee \ (x_1 = 1 \wedge x_2 = 0) \equiv (x_1 + x_2 \leq 1)$$

- The number of places involved in each invariant cannot be bounded a priori. In some Petri nets, all variables may participate in the generated invariants.
- The transitions of the Petri Net may consume and produce a number of tokens which is greater than one. In the nets where this happens, invariants tend to contain non-unit coefficients.

Intuitively, individual invariants can be encoded accurately as linear inequalities and equalities. As a set of invariants may include conjunctions and disjunctions of these individual invariants, Presburger arithmetics seems the right choice for this class of problems. However, using Presburger in this problem is difficult because of the high complexity, e.g. [81]. Defining a customized abstract domain which is specially adapted to this problem seems to be a more promising approach.

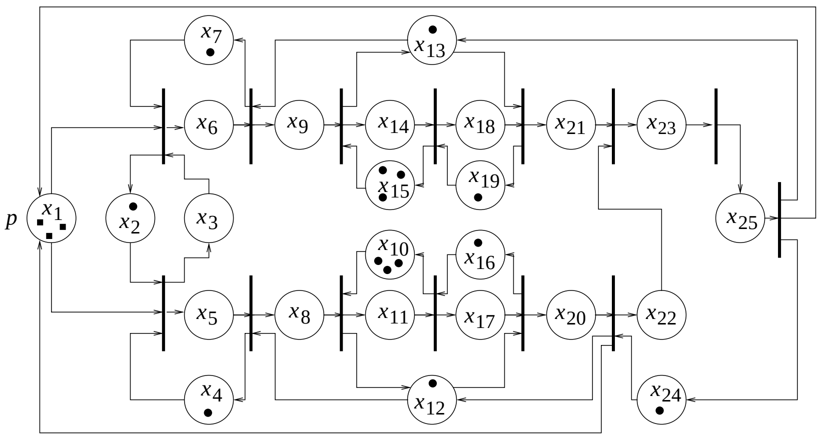


Figure 6.1: Petri Net model of an automated manufacturing system

6.4.3 Experiments

We have evaluated the use of two abstract domains in this problem: convex polyhedra and polynomial equalities [137]. We will briefly report our experiences in one example using the convex polyhedron abstract domain.

Figure 6.1 shows a Petri Net model of an automated manufacturing system [162]. This manufacturing system consists of several elements: four machines ($M_1 - M_4$), two robots $(R_1 - R_2)$, two buffers with capacity 3 $(B_1 - B_2)$ and an assembly cell. The place x_1 models the entry point for raw material, while the place x_{10} (x_{15}) represents the availability of the buffer B_1 (B_2). The place x_{12} (x_{13}) models the availability of the robot R_1 (R_2), whereas the place x_{25} represents the delivery point for the final product. Finally, the places x_4, x_7, x_{16} and x_{19} model the availability of the machines M_1 to M_4.

The initial marking of this Petri Net is as follows. The entry point x_1 has an undetermined number of tokens p, as we want to study the behavior of the system depending on the quantity of available raw materials. The capacities of the buffers, x_{10} and x_{15}, have 3 tokens as the buffers have size 3. Finally, places x_2, x_4, x_7, x_{12}, x_{13}, x_{16}, x_{19} and x_{24} have one token, and the rest of places have no tokens in the initial marking.

Some relevant properties in this system are *boundedness* and *liveness*, i.e. deadlock freedom. In previous work, these properties have been studied in detail. In [162], it was proven that the system is bounded, and that it is live only for some values of p, namely $2 \leq p \leq 4$. A different approach based on integer programming [46] managed to prove liveness for a wider interval of values, $1 \leq p \leq 8$. Also, a sequence of firings leading to a deadlock when $p > 8$ was shown. Later work has revisited these results using other techniques such as Presburger arithmetics [81], real arithmetics [24] and inductive linear inequalities based on Farkas' lemma [140]. Compared to these methods, abstract interpretation analysis provides several advantages: the guarantee of termination provided by the widening operator, the acceleration of the converge also due to the widening, and the extensibility with new abstract domains adapted to the particular problem.

An analysis based on convex polyhedra reveals the following set of invariants. Equality and inequality invariants are listed separately:

$$\begin{array}{ll}
x_{18} + x_{19} = 1 & x_2 + x_3 = 1 \\
x_8 + x_{12} + x_{20} = 1 & x_4 + x_5 = 1 \\
x_{22} + x_{23} + x_{24} + x_{25} = 1 & x_6 + x_7 = 1 \\
x_9 + x_{13} + x_{21} + x_{23} + x_{25} = 1 & x_{10} + x_{11} = 3 \\
x_{19} + x_{17} + x_{15} + x_{13} + x_{11} + x_7 + x_5 + x_2 - x_{24} - x_{12} = 5 & x_{14} + x_{15} = 3 \\
x_{19} + x_{17} + x_{15} + x_{13} + x_{11} + x_7 + x_5 - x_{24} - x_{12} - x_3 = 4 & x_{16} + x_{17} = 1 \\
x_{19} + x_{15} + x_{13} + x_{12} + x_7 + p - x_{17} - x_{11} - x_5 - x_1 = 7 & \\
\hline
x_{25} + x_{24} + x_{23} \leq 1 & x_7 \leq 1 \\
x_{25} + x_{23} + x_{21} + x_{13} \leq 1 & x_{15} \leq 3 \\
x_{19} + x_{15} + x_{13} + x_7 - x_{24} - x_1 \leq 5 & x_{19} \leq 1 \\
x_{19} + x_{17} + x_{15} + x_{13} + x_{11} + x_7 + x_5 - x_{24} - x_{12} \geq 4 & x_{20} + x_{12} \leq 1
\end{array}$$

These invariants are sufficient to prove that the system is bounded. Nevertheless, these invariants are incapable of proving deadlock freedom, although it can be proven with the polynomial equality abstract domain. The reason is the inability to capture disjunctions of invariants in convex polyhedra. The same problem can be observed in other Petri Net models. Further work is required to find an abstract domain which is fully appropriate for the reachability problem. A promising research direction is a customized version of the *finite power-set* of convex polyhedra, i.e. an abstract domain defined as a disjunction of several convex polyhedra. The general definition of this domain fails to take full advantage of the characteristics of the problem, so additional work is required to adapt the domain for this specific problem.

6.5 Conclusions

We have described a set of future research directions arising from the work presented in this thesis. Improving the scalability of the proposed method, together with establishing new properties and abstract operators for the octahedron abstract domain seem the most promising at this point. Direct applications of octahedra to static analysis problems with numerical constraints are also possible.

Finally, other problems of a similar nature to the verification of timed circuits will also be approached with abstract interpretation techniques. A case study for the analysis of the reachable markings in Petri Nets has been described. Preliminary results highlight the potential of abstract interpretation in this problem, and point out the necessity of relational numerical abstract domains which are more efficient than convex polyhedra for more specific problems.

Chapter 7

Conclusions

This thesis has presented a set of contributions in two areas: the verification of timed circuits and the framework of abstract interpretation. This chapter analyzes the work contained in the previous chapters, and draws several conclusions derived from the results.

Timed Circuits and the Verification Problem

The verification of a timed system is a complex problem. Encoding time, which is an integral magnitude with potentially an infinite number of values, is more difficult than encoding boolean or discrete variables. An interesting class of timed systems to be considered is that of *timed circuits*. In these circuits, the dynamic behavior can be described with two types of transitions: discrete and timed. The discrete transitions describe the logic function implemented by the gates, the connection among the different gates of the netlist, and the interaction with the environment. Meanwhile, the timed transitions model the delay of gates, wires and environment events. As these circuits are typically highly concurrent, verification combines the complexities of concurrency and time.

A collection of verification techniques called *metric timing* use techniques adapted from those in timed systems. Scalability is an issue in metric timing, because the number of clocks in the model increases quickly with the size of the circuit. Another strategy consists in studying the untimed behavior of the system, limiting the timing information to constraints on the relative order of events. These approaches, such as *relative timing* and *chain constraints*, can take advantage of the efficient BDD methods available for manipulating untimed state spaces. In principle, this allows a better scalability to relative timing methods versus metric timing.

Adding Parameters to the Picture

In regular timed systems, the temporal characteristics of the system, such as delays and the bounds for clocks, are *metric*: they are defined with a constant value or an interval of constants. It is also possible to study a *parametric* version of the same problem, where some of the values are symbols that become parameters of the problem. The verification of parametric timed systems is more complex than their metric counterparts. For instance, many problems are undecidable to begin with. The parametric regions cannot be encoded using efficient data structures such as difference bound matrices (DBM), with a worst-case polynomial complexity on the number of clocks. Instead, methods with exponential complexity like convex polyhedra or Presburger arithmetic, or methods without a guarantee of termination must be employed. In all these cases, complexity is extremely sensitive to the number of parameters that appear in the problem, in addition to the number of clocks.

A parametric version of the verification of timed circuits can also be defined. Abstract interpretation is an adequate technique to study this problem because of several reasons. First, the termination of the analysis is guaranteed. Also, abstract interpretation is well-suited for the discovery of numerical properties. There are several numerical abstract domains which provide different trade-offs between precision and efficiency. Moreover, it is possible to define abstract domains which are targeted at solving a specific problem very efficiently.

Verification of Timed Circuits with Symbolic Delays

The presented work describes a methodology for the verification of timed circuits using symbolic delays. In this method, the output timing constraints are a system of linear inequalities over the parameters. In terms of execution time and memory usage, other methods based on metric timing, relative timing and chain constraints are more efficient than the verification with symbolic delays. This means that these methods can scale up to larger circuits. Even then, the benefits of parametric timing constraints balance this problem.

With respect to metric constraints, parametric timing constraints offer several advantages. First, this class of timing constraints is *technology independent*: it is characterizing the behavior of the circuit for any possible delay of the elements. In this sense, it is superior to metric timing techniques, where the analysis can only check the correctness for the specific set of delays used as input. Furthermore, parametric timing constraints can be used as the *guidelines to select delays*, e.g. aiding to reduce the latency of a circuit while ensuring correctness. Another advantage is the *simple and efficient procedure to validate the timing constraints*: if the inequality evaluates to true with the chosen constant delay, then the constraint is satisfied. In this respect, these constraints are superior to relative timing constraints, where validation is not trivial. Finally, parametric timing constraints provide a *meaningful feedback to the designer* as they identify competing delay paths within the circuit.

Other advantages are related to the method used to compute the parametric timing constraints. Firstly, *known delays can be included in the formulation of the problem*, such as delays defined by a standard or by the specification. Replacing parameters by known constants reduces the complexity of the verification. Moreover, the algorithm is *fully automatic*, with no user interaction required to obtain the timing constraints. Also, as the timing constraints are encoded using convex polyhedra, it is simple to *detect redundant constraints or unsatisfiable sets of constraints*. These observations favor the method presented in this thesis with respect to the chain constraint approach.

The proposed methodology also exhibits several favorable traits in comparison with previous methods based on parametric timing constraints. Drawbacks such as the *lack of full automation*, the *dependence on metric timing constraints* at some level or the *inability to analyze sequential circuits* are solved by the methods described in Chapter 4. Furthermore, our approach is able to verify circuits with more symbolic delays than previous methods.

Given all these considerations, the verification with symbolic delays offers relevant benefits in the analysis of small controllers, either designed by hand or by sophisticated synthesis tools, whose behavior depends on the timing characteristics of the components. Several asynchronous controllers have been used to illustrate this point. Regarding the scalability of the approach, extensions such as compositional verification methods or automatic abstraction techniques can be explored to improve the scalability of the approach.

Numerical Abstract Domains

Another strategy to improve the efficiency of the verification procedure relies on adopting a more efficient numerical abstract domain to represent the timing constraints. A study of the type of numerical properties captured as timing constraints reveals that most constraints are unit inequalities, i.e. all coefficients of the linear inequalities are -1, 0 or $+1$. This information allows more efficient abstract domains for this specific problem. The fact that all the variables involved in the constraints are non-negative permits further optimizations.

Thinking purely in terms of complexity, this restriction does not improve the situation drastically. Given a system with n variables, there is still an exponential number of possible unit inequalities ($3^n - 1$). Therefore, the operations will have a worst-case exponential cost, even if it is an exponential of a lower order than that of convex polyhedra operations. However, experimental results have shown that the improvement is noticeable and worthwhile.

We have proposed two implementations for this abstract domain named octahedron. These implementations take advantage of the fact that coefficients are discrete by applying techniques from discrete domains, such as decision diagrams and bit-vectors. The major benefit of both approaches is an important reduction in memory usage with respect to convex polyhedra. In the case of decision diagrams, the cost of this reduction is an increase in execution time, which is spent minimizing the representation. On the other hand, bit-vectors improve both memory and execution time, although the reduction in terms of memory is smaller in large examples than that of decision diagrams. Regarding precision, the experimental results have shown that the difference in precision between convex polyhedra and octahedra is negligible in the problem of verifying a timed circuit.

These techniques allow the verification of larger systems than what is possible with convex polyhedra. Remarkably, the assumptions used to define octahedra also occur in several static analysis problems. Therefore, the improvements achieved in this area may be exported to other analysis problems.

There are several directions of future research in this field. On one hand, a more efficient algorithm for the computation of the canonical form of octahedra would improve the precision of the OhDD implementation, and potentially also accelerate the operations. Another focus of future research is the definition of exact versions of the operators of octahedra which are approximate, like in the bit-vector implementation. Finally, other numerical abstract domains which are appropriate for the timing verification problem can be explored.

Bibliography

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *Proc. International Symposium on Static Analysis*, pages 52–66. Springer-Verlag, 1996.

[2] R. Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford University, Aug. 1991.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, pages 3–34, 1995.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] R. Alur, K. Etessami, S. L. Torre, and D. Peled. Parametric temporal logic for “model measuring”. *ACM Trans. Comput. Logic*, 2(3):388–407, 2001.

[6] R. Alur and T. Henzinger. Logics and Models of Real-Time: A Survey. In *Real Time: Theory in Practice*, volume 600, pages 74–106. Springer-Verlag, 1992.

[7] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88:971–984, 2000.

[8] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *ACM Symposium on Theory of Computing*, pages 592–601, 1993.

[9] T. Amon and G. Borriello. An approach to symbolic timing verification. In *Proc. ACM/IEEE Design Automation Conference*, pages 410–413. IEEE Computer Society Press, 1992.

[10] T. Amon, G. Borriello, T. Hu, and J. Liu. Symbolic timing verification of timing diagrams using Presburger formulas. In *Proc. ACM/IEEE Design Automation Conference*, pages 226–231, jun 1997.

[11] T. Amon and H. Hulgaard. Symbolic time separation of events. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 83–93, 1999.

[12] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proc. International Conference on Computer Aided Verification*, pages 419–434, 2000.

[13] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. In *Computer Aided Verification*, pages 368–372, 2001.

[14] A. Arnold. *Finite Transition Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[15] E. Asarin, O. Maler, and A. Pnueli. On discretization of delays in timed automata and digital circuits. In *International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 470–484. Springer-Verlag, 1998.

[16] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In R. Cousot, editor, *Proc. International Symposium on Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 337–354. Springer-Verlag, 2003.

[17] R. Bagnara, P. M. Hill, E. Ricci, E. Zaffanella, C. Medori, and A. Zaccagnini. PPL: The Parma Polyhedra Library. http://www.cs.unipr.it/ppl/.

[18] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL implementation secrets. In *Proc. of 7*th *International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 3–22, 2002.

[19] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using Clock Difference Diagrams. In *Proc. International Conference on Computer Aided Verification*, pages 341–353, 1999.

[20] W. Belluomini and C. Myers. Timed state space exploration using POSETs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(5), 2000.

[21] W. J. Belluomini and C. J. Myers. Timed circuit verification using TEL structures. *IEEE Transactions on Computers*, 20(1):129–146, 2001.

[22] J. Bengtsson. memtime - utility for the collection of memory and CPU time usage information. Available online at `www.update.uu.se/~johanb/memtime/`.

[23] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Proc. Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1995.

[24] B. Bérard and L. Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *Proc.Int. Conf. on Concurrency Theory*, volume 1664, pages 178–193. Lecture Notes in Computer Science, Aug. 1999.

[25] D. Beyer. Improvements in BDD-based reachability analysis of timed automata. In *Proc. International Symposium of Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 318–343. Springer-Verlag, 2001.

[26] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566, pages 85–108. Springer-Verlag, Oct. 2002.

[27] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriu. Handshake protocols for desynchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2004.

[28] A. Bouajjani, A. Collomb-Annichini, Y. Lakhnech, and M. Sighireanu. Analyzing fair parametric extended automata. In *Proc. International Symposium on Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 335–355, 2001.

[29] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *Proc. of the International Workshop on Programming Languages Implementation and Logic Programming PLILP'90*, volume 456, pages 307–323. Lecture Notes in Computer Science, 1990.

[30] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735:128–141, 1993.

[31] O. Bournez and O. Maler. On the representation of timed polyhedra. In *Automata, Languages and Programming*, pages 793–807, 2000.

[32] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1998.

[33] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proc. International Conference on Computer Aided Verification*, volume 1427, pages 546–550. Springer-Verlag, 1998.

[34] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Proceedings of the 9th In-*

ternational Conference on Computer Aided Verification, pages 179–190. Springer-Verlag, 1997.

[35] M. Bozga, O. Maler, and S. Tripakis. Efficient verification of timed automata using dense and discrete time semantics. In *Proc. Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 125–141, 1999.

[36] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. ACM/IEEE Design Automation Conference*, pages 40–45. ACM Press, 1990.

[37] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[38] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *Proc. International Conference on Computer Aided Verification*, pages 400–411, 1997.

[39] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999.

[40] S. Chakraborty. *Polynomial-Time Techniques for Approximate Timing Analysis of Asynchronous Systems*. PhD thesis, Stanford University, Aug. 1998.

[41] S. Chakraborty and D. L. Dill. Approximate algorithms for time separation of events. In *Proc. International Conf. Computer-Aided Design (ICCAD)*. IEEE Computer Society Press, 1997.

[42] S. Chakraborty, D. L. Dill, and K. Y. Yun. Min-max timing analysis and an application to asynchronous circuits. *Proceedings of the IEEE*, 87(2):332–346, 1999.

[43] C. Chen, T. Lin, and H. Yen. Modelling and analysis of asynchronous circuits and timing diagrams using parametric timed automata. In *Proc. International Conf. on Modelling, Identification and Control*, pages 500–505, Feb. 2004.

[44] L. Chen, L. Harrison, and K. Yi. Efficient computation of fixpoints that arise in complex program analysis. *Journal of Programming Languages*, 3(1):31–68, 1995.

[45] N. Chernikova. Algoritm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 6(8):282–293, 1964.

[46] F. Chu and X.-L. Xie. Deadlock analysis of Petri nets using siphons and mathematical programming. *IEEE Transactions on Robotics and Automation*, 13(6):793–804, Dec. 1997.

[47] T.-A. Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, MIT, June 1987.

[48] R. Clarisó and J. Cortadella. Symbolic timing analysis for the verification of asynchronous circuits. In *Handouts of the Asynchronous Circuit Design (ACID) Workshop*, Jan. 2003.

[49] R. Clarisó and J. Cortadella. Verification of timed circuits with symbolic delays. In *International Workshop on Logic Synthesis (IWLS)*, pages 310–317, May 2003.

[50] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *Proc. International Symposium on Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag, Aug. 2004.

[51] R. Clarisó and J. Cortadella. Verification of parametric timed circuits using octahedra. In *Proc. Int. Workshop on Designing Correct Circuits (DCC'05)*, Mar. 2004.

[52] R. Clarisó and J. Cortadella. Verification of timed circuits with symbolic delays. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 628–633, Jan. 2004.

[53] R. Clarisó and J. Cortadella. Verification of concurrent systems with parametric delays using octahedra. In *Proc. Int. Conf. on Application of Concurrency to System Design*, pages 122–131, 2005.

[54] R. Clarisó, E. Rodríguez-Carbonell, and J. Cortadella. Derivation of non-structural invariants of Petri nets using abstract interpretation. In *Proc. Int. Conf. On Application and Theory of Petri Nets and Other Models of Concurrency*, volume 3536 of *Lecture Notes in Computer Science*, pages 188–207. Springer-Verlag, 2005.

[55] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proc. of the 4th Annual Symposium on Logic in computer science*, pages 353–362. IEEE Press, 1989.

[56] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[57] C. Colby. Determining storage properties of sequential and concurrent programs with assignment and structured data. In *Proc. International Symposium on Static Analysis*, pages 64–81, 1995.

[58] J. Cortadella, M. Kishinevsky, S. M. Burns, A. Kondratyev, L. Lavagno, K. S. Stevens, A. Taubin, and A. Yakovlev. Lazy transition systems and asynchronous circuit synthesis with relative timing assumptions. *IEEE Transactions on Computer-Aided Design*, 21(2):109–130, Feb. 2002.

[59] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[60] P. Cousot. Abstract interpretation: Achievements and perspectives. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*. Scuola Superiore G. Reiss Romoli, July 2000.

[61] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, Berlin, Germany, 2001.

[62] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[63] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[64] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.

[65] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295, Berlin, Germany, 1992. Springer-Verlag.

[66] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, New York, 1978.

[67] G. Dantzig and B. Eaves. Fourier-Motzkin elimination and its dual. *Journal of combinatorial theory*, 14:288–297, 1973.

[68] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proc. IEEE Real-Time Systems Symposium*, pages 73–81. IEEE Computer Society Press, Dec. 1996.

[69] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[70] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Verification of asynchronous interface circuits with bounded wire delays. *Journal of VLSI Signal Processing*, 7(1/2):161–182, Feb. 1994.

[71] D. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In *Proc. International Conference on Computer Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[72] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 197–212. Springer-Verlag, 1989.

[73] A. Dolzmann, T. Sturm, and V. Weispfenning. A new approach for automatic theorem proving in real geometry. *Journal of Automated Reasoning*, 21(3):357–380, 1998.

[74] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming lan guage design and implementation*, pages 155–167. ACM Press, 2003.

[75] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, M. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In O. Maler, editor, *Hybrid and Real-Time Systems*, pages 346–360, Grenoble, France, 1997. Springer Verlag, LNCS 1201.

[76] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.

[77] E. A. Emerson and E. M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[78] E. A. Emerson and R. J. Trefler. Parametric quantitative temporal reasoning. In *Logic in Computer Science*, pages 336–343, 1999.

[79] F. Fernández and P. Quinton. Extension of Chernikova's algorithm for solving general mixed linear programming problems. Technical Report 437, IRISA, Rennes, France, 1988.

[80] M. J. Fisher and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. In *Proc. SIAM-AMS*, volume 7, pages 27–41, 1974.

[81] L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetics. In *Proc.Int. Conf. on Concurrency Theory*, volume 1243, pages 213–227. Lecture Notes in Computer Science, July 1997.

[82] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[83] A. J. Gerber. Process synchronization by counter variables. *SIGOPS Operating Systems Review*, 11(4):6–17, 1977.

[84] A. Gollu, A. Puri, and P. Varaiya. Discretization of timed automata. In *Proc. IEEE Conference on Decision and Control*, pages 957–958, 1994.

[85] P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[86] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proc. of the Fourth International Joint Conference on the Theory and Practice of Software Development (TAPSOFT '91)*, volume 493, pages 169–192. Springer-Verlag, apr 1991.

[87] A. P. Gupta and D. P. Siewiorek. Automated multi-cycle symbolic timing verification of microprocessor-based designs. In *Proc. ACM/IEEE Design Automation Conference*, pages 113–119. ACM Press, 1994.

[88] R. Gupta. Generalized dominators and post-dominators. In *Proc. of the ACM Symposium on Principles of Programming Languages*, volume 19, pages 246–257, Jan. 1992.

[89] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *Proc. International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346, Elounda, Greece, 1993. Springer-Verlag.

[90] N. Halbwachs, D. Merchat, and C. Parent-Vigouroux. Cartesian factoring of polyhedra in linear relation analysis. In *Proc. International Symposium on Static Analysis*, pages 355–365. LNCS 2694, Springer Verlag, June 2003.

[91] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[92] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Proc. REX Workshop Real-Time: Theory in Practice*, volume 600, pages 226–251. LNCS, New York, 1992.

[93] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proc. Int. Colloquium on Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer-Verlag, 1992.

[94] H. Hulgaard. *Timing Analysis and Verification of Timed Asynchronous Circuits*. PhD thesis, Department of Computer Science, University of Washington, 1995.

[95] H. Hulgaard and T. Amon. Symbolic timing analysis of asynchronous systems. *IEEE Transactions on Computer-Aided Design*, 19(10):1093–1104, Oct. 2000.

[96] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Transactions on Computers*, 44(11):1306–1317, Nov. 1995.

[97] T. Hune, J. Romijn, M. Stoelinga, and F. W. Vaandrager. Linear parametric model checking of timed automata. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 189–203, 2001.

[98] N. Ishiura, M. Takahashi, and S. Yajim. Time-symbolic simulation for accurate timing verification of asynchronous behavior of logic circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 497–502, 1989.

[99] B. Jeannet. New Polka: Convex Polyhedra Library. Available online at `http://www.irisa.fr/prive/bjeannet/newpolka.html`.

[100] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[101] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.

[102] H. Kim, P. A. Beerel, and K. Stevens. Relative timing based verification of timed circuits and systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 115–124, Apr. 2002.

[103] X. Kong and R. Negulescu. Bolstering faith in GasP circuits through formal verification. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 113–124. IEEE Computer Society Press, Apr. 2004.

[104] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state space reduction. In *Proc. IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, Dec. 1997.

[105] L. Lavagno, K. Keutzer, and A. L. Sangiovanni-Vincentelli. Synthesis of hazard-free asynchronous circuits with bounded wire delays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1), 1995.

[106] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vincentelli. An efficient heuristic procedure for solving the state assignment problem for event-based specifications. *IEEE Transactions on Computer-Aided Design*, 14(1):45–60, Jan. 1995.

[107] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[108] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In *Proc. Correct Hardware Design and Verification Methods (CHARME)*, pages 189–205, Oct. 1995.

[109] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *Proc. ACM Int. Conf. on Supercomuting*, pages 226–235, 1992.

[110] I. Mastroeni. Numerical power analysis. In *Proc. Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 117–137. Springer-Verlag, 2001.

[111] C. Mauras. Symbolic simulation of interpreted automata. In *3rd Workshop on Synchronous Programming*, Dec. 1996.

[112] S. Minato. Zero-supressed BDDs for set manipulation in combinatorial problems. In *Proc. ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

[113] A. Miné. A new numerical abstract domain based on Difference-Bound Matrices. In *Programs as Data Objects II*, volume 2053 of *LNCS*, pages 155–172. Springer-Verlag, May 2001.

[114] A. Miné. The octagon abstract domain. In *Analysis, Slicing and Tranformation (in Working Conference on Reverse Engineering)*, IEEE, pages 310–319. IEEE CS Press, Oct. 2001.

[115] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Proceedings 13th International Workshop on Computer*

Science Logic, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, Sept. 1999.

[116] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. ACM/IEEE Design Automation Conference*, pages 530–535, June 2001.

[117] M. Müller-Olm and H. Seidl. Computing polynomial program invariants. *Information Processing Letters (IPL)*, 91(5):233–244, 2004.

[118] T. Murata. State equation, controllability and maximal matchings of Petri nets. *IEEE Transactions on Automatic Control*, AC-22(3):412–416, 1977.

[119] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989.

[120] C. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 279–282. IEEE Computer Society Press, Oct. 1992.

[121] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Electrical Engineering, Stanford University, Oct. 1995.

[122] C. J. Myers, W. Belluomini, K. Killpack, E. M. er, E. Peskin, and H. Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, 2001.

[123] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[124] R. Negulescu. A technique for finding and verifying speed-dependences in gate circuits. Research Report CS-97-28, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, Aug. 1997.

[125] R. Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits*. PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, Aug. 1998.

[126] R. Negulescu and A. Peeters. Verification of speed-dependences in single-rail handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 159–170, 1998.

[127] C. A. Nelson, C. J. Myers, and T. Yoneda. Efficient verification of hazard-freedom in gate-level timed asynchronous circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 424–431, Nov. 2003.

[128] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[129] N.Ishiura, Y.Deguchi, and S.Yajima. Coded time-symbolic simulation using shared binary decision diagram. In *Proc. ACM/IEEE Design Automation Conference*, pages 130–135, 1990.

[130] The Omega project: Frameworks and algorithms for the analysis and transformation of scientific programs. Available online at `http://www.cs.umd.edu/projects/omega/omega.html`.

[131] D. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, July 1978.

[132] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, *Proc. Int. Conf. on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435. Springer-Verlag, June 1994.

[133] M. A. Peña, J. Cortadella, A. Kondratyev, and E. Pastor. Formal verification of safety properties in timed circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11. IEEE Computer Society Press, Apr. 2000.

[134] C. Piguet et al. Memory element of the Master-Slave latch type, constructed by CMOS technology. US Patent 5,748,522, 1998.

[135] A. Pnueli. The temporal logic of programs. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 45–57, 1977.

[136] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, (8):102–104, Aug. 1992.

[137] E. Rodríguez-Carbonell and D. Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *Proc. International Symposium on Static Analysis*, pages 280–295. Springer-Verlag, Aug. 2004.

[138] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: An asynchronous instruction length decoder. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70, Apr. 1999.

[139] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 42–47, 1993.

[140] S. Sankaranarayanan, H. Sipma, and Z. Manna. Petri net analysis using invariant generation. In *Verification: Theory and Practice*, pages 682–701. Springer-Verlag, 2003.

[141] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. ACM SIGPLAN Principles of Programming Languages*, pages 318–329, 2004.

[142] S. A. Seshia, R. E. Bryant, and K. S. Stevens. Modeling and verifying circuits using Generalized Relative Timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 98–108. IEEE Computer Society Press, 2005.

[143] M. Silva, E. Teruel, and J. M. Colom. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:309–373, 1998.

[144] A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In M. Leuschel, editor, *Proceedings of Logic Based Program Development and Transformation*, LNCS 2664, pages 71–89. Springer-Verlag, 2002.

[145] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes (extended abstract). In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 216–226. ACM Press, 1991.

[146] F. Somenzi. CUDD: Colorado university decision diagram package. Available online at `http://vlsi.colorado.edu/˜fabio/CUDD`.

[147] R. F. L. Spelberg and W. Toetenel. Real-time model checkin based on splitting. In *Proc. International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 143–157, 1998.

[148] K. Stevens, R. Ginosar, and S. Rotem. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, Apr. 1999.

[149] I. Sutherland and S. Fairbanks. GasP: A minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53, 2001.

[150] R. Thacker, W. Belluomini, and C. J. Myers. Timed circuit synthesis using implicit methods. In *Proc. International Conference on VLSI Design*, pages 181–188, 1999.

[151] A. van Gelder. Deriving constraints among argument sizes in logic programs (extended abstract). In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 47–60. ACM Press, 1990.

[152] H. L. Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, Rennes, France, 1992.

[153] F. Wang. Efficient data structure for fully symbolic verification of real-time software systems. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 157–171, 2000.

[154] F. Wang. Parametric analysis of computer systems. *Formal Methods in System Design*, 17:39–60, 2000.

[155] F. Wang. Symbolic verification of complex real-time systems with Clock-Restriction Diagrams. In *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems*, pages 232–250. Kluwer Academic Publishers, 2001.

[156] F. Wang. Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305, Aug. 2004.

[157] F. Wang. Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. In *Proc. International Conference on Computer Aided Verification*. Springer-Verlag, July 2004.

[158] T. Yoneda, T. Kitai, and C. Myers. Automatic derivation of timing constraints by failure analysis. In *Proc. International Conference on Computer Aided Verification*, pages 195–208, 2002.

[159] T. Yoneda and H. Ryu. Timed trace theoretic verification using partial order reduction. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108–121, Apr. 1999.

[160] H. Zheng. *Modular Synthesis and Verification of Timed Circuits Using Automatic Abstraction*. PhD thesis, The University of Utah, Aug. 2001.

[161] H. Zheng, C. J. Myers, D. Walter, S. Little, and T. Yoneda. Verification of timed circuits with failure directed abstractions. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 28–35, Nov. 2003.

[162] M. Zhou, F. DiCesare, and A. Desrochers. A hybrid methodology for synthesis of Petri net models for manufacturing systems. *IEEE Transactions on Robotics and Automation*, 8(3):350–361, June 1992.

Appendix A

OhDD Algorithms

This Appendix presents the decision diagram implementation of several operators of the octahedron abstract domain. It is provided to better illustrate the concepts described in Section 5.3.

Decision Diagram Packages

The following is a brief description of the internal structure of a decision diagram package. Only the minimum details required to understand the algorithms are discussed. The interested reader can find additional information in [36, 146].

A decision diagram is a directed acyclic graph (DAG). The basic data structure of this graph is the *node*, which can be a *constant node* or a *decision node*. Constant nodes store the possible values of the function encoded in the decision diagram. On the other hand, each decision node is labelled with a decision atom, e.g. a boolean variable, and the children nodes are chosen according to the values of the atom. Throughout this thesis, several classes of decision diagrams have been presented, with different types of decision atoms, such as differences of pairs of expressions (DDD) or or linear inequalities (DDC). In any case, it is important that there exists a *total order* among the atoms. This order defines the position of each atom in the path from the root node to the constant nodes at the bottom of the diagram. In other words, the order ensures that all the nodes at the same level are labeled with the same decision atom.

The ordering of atoms has a large impact in the size of the decision diagram, and therefore, in the effectivity of the algorithms. A strategy called *dynamic reordering* [139] can be used to choose the ordering that minimizes a decision diagram at a given point of the execution. However, this technique is outside the scope of this description, and thus we will assume a static order.

Several decision diagrams can be stored at once in the same package. As each decision diagram is a DAG with a single root, it can be identified by the root node.

The canonicity of the decision diagrams is ensured by a set of reduction rules. Although each diagram has different rules, a typical rule is the merging of isomorphic subgraphs. This reduction rule is usually implemented using a hash-table known as the *unique table*. Whenever a node needs to be created, the creation is implemented by a call to:

DD_UniqueNode($atom$, $childrenlist$)

which returns a new node if an isomorphic node does not exist, or the isomorphic node if it is already allocated inside the package. The implementation of this procedure takes advantage of the fact that each children in the $childrenlist$ is also canonical. Using this method, it is possible to guarantee that any pair of atom + list of children is encoded by a single node.

All the operations, such as the intersection of the union, can be implemented as recursive procedures. These procedures receive as arguments the root nodes of the decision diagrams. A divide and conquer approach is used to implement the operations: the current call performs the operation on the top atom, leaving the remaining work (applying the operation to the children) to recursive calls on the children. However, as decision diagrams are DAGs, it is possible that many recursive calls have the same arguments. In order to avoid repeated work, decision diagram packages implement a *cache* memory that stores recent operations. A call of the form

DD_CacheLookup($operation$, $argumentlist$)

checks whether the operation is already available in the cache memory and therefore, does not need to be recomputed. Similarly, after any call, the procedure should store the result in the cache memory with a method like:

DD_CacheInsert($operation$, $argumentlist$, $result$)

In addition to the methods described so far, decision diagram packages provide several methods to manipulate nodes. At least the following methods should be provided (the names used in the pseudocode appear between parenthesis)

- Testing whether a node is a constant or a decision node (DD_IsConstant). This method is used by recursive procedures to detect when they have reached the bottom of the diagram.
- Getting the children of a given node (DD_NegArc, DD_ZeroArc, DD_PosArc).
- Choosing the top atom of several decision diagrams according to the order (DD_TopVariable). In each recursive call, only the top atom is processed.

Reduction rules

In order to implement the reduction rules, two basic operations should be defined. First, how to obtain the OhDD for the three cofactors (coefficients) of a given variable. And second, how to build a new OhDD from the three cofactors of a given variable. Figure A.1 shows the pseudocode for these two procedures, called *DD_GetCofactors* and *DD_CombineCofactors*.

The only remarkable aspect of *DD_GetCofactors* is how to compute the cofactors of a variable that does not appear in a OhDD. If a variable is missing, it means that it has been reduced. In a OhDD with non-negative variables, this means that its negative cofactor is $-\infty$ while its positive and zero cofactor are equal to the OhDD (the reduction rule is described in Figure 5.8). A similar implementation produces the reduction rules for unconstrained variables.

Saturation

The saturation procedure can be implemented symbolically. Instead of choosing two constraints and computing its linear combination, the linear combination of a whole set of constraints is computed in one step. Figures A.2 and A.3 show the pseudocode that performs this saturation.

The recursive saturation algorithm *SaturateRecur* computes the linear combination of its two parameters. Intuitively, the computation is split according to the top variable of the decision diagram. The only cases relevant to our computation are those where the top variable has a coefficient in $\{-1, 0, +1\}$. For example, the linear combination has a coefficient $+1$ if one of the arguments has coefficient 0 and the other has coefficient $+1$. The remaining variables of the linear combination are computed recursively using the same algorithm.

Saturation is performed by computing these linear combinations and adding them to the system of inequalities until a fixpoint is reached. This computation is described in the procedure *Saturate* in Figure A.2. Notice that after computing each set of linear combinations, they are added to the OhDD using the maximum operator, which in saturated OhDD corresponds to the intersection.

Other operations

The intersection of two octahedra has been defined as the union of the sets of constraints of both octahedra, choosing the maximum constant for those constraints that appear in both octahedra. In a OhDD, constraints that do not appear in an octahedron are represented by the terminal $-\infty$, e.g. $(x+y-z \geq -\infty)$. Therefore, the intersection of OhDD can be implemented by taking the *maximum* of the two arguments for each path between the root and the terminal nodes. The pseudocode that computes this maximum is shown in Fig. A.4. The result of this operation is not necessarily saturated.

The same concept can be applied to the union of octahedra. The union can be computed as the *minimum* of the two arguments for each path between the root and the terminal nodes.

Function *DD_GetCofactors*(f, var)
Input: An OhDD f over non-negative variables and a variable var. If var appears in the decision diagram f, it must appear as the top variable.
Output: The 3 cofactors of f for variable var, $< f^-, f^0, f^+ >$.

```
if DD_IsConstant($f$) $\vee$ DD_TopVariable($f$) $\neq$ var then
   $< f^-, f^0, f^+ >$ := $< -\infty, f, f >$
else
   $< f^-, f^0, f^+ >$ := $<$ DD_NegArc($f$), DD_ZeroArc($f$), DD_PosArc(f) $>$
endif
return $< f^-, f^0, f^+ >$
```

Function *DD_CombineCofactors*(var, f^-, f^0, f^+)
Input: The three cofactors of a OhDD for a given variable var. Any variable in the cofactors must appear after var in the ordering.
Output: A OhDD where $< f^-, f^0, f^+ >$ are the three cofactors for variable var.

```
if $f^0 = f^+ \wedge f^- = -\infty$ then
   return $f^0$
else
   return DD_UniqueNode($var$, $f^-$, $f^0$, $f^+$)
endif
```

Figure A.1: Pseudocode for the methods that implement the reduction of zero coefficients with non-negative variables.

Function *Saturate*(f)
Input: A OhDD f.
Output: The saturation of the OhDD f.

```
do
   old := $f$
   res := SaturateRecur($f$, $f$)
   $f$ := MaximumRecur($f$, res)
while $f \neq$ old
return res
```

Figure A.2: Pseudocode for the saturation procedure in the OhDD implementation.

Function *SaturateRecur*(f, g)
Input: Two OhDD called f and g.
Output: The OhDD describing the linear combination of f and g, ignoring constraints with a coefficient outside $\{-1, 0, +1\}$.

```
{ Terminal cases }
if DD_IsConstant($f$) $\wedge$ DD_IsConstant($g$) then
   return DD_Sum($f$, $g$)
endif
if $f = +\infty \vee g = +\infty$ then return $+\infty$ endif
if $f = -\infty \vee g = -\infty$ then return $-\infty$ endif

{ Lookup the result in the cache }
res := DD_CacheLookup(SaturateRecur, $f$, $g$)
if res $\neq$ null then return res endif
top := DD_TopVariable($f$, $g$)
$< f^-, f^0, f^+ >$ := DD_GetCofactors(top, $f$)
$< g^-, g^0, g^+ >$ := DD_GetCofactors(top, $g$)

{ Recursive calls for top coefficient $= 0$ }
call1 := SaturateRecur($f^0$, $g^0$)
call2 := SaturateRecur($f^+$, $g^-$)
call3 := SaturateRecur($f^-$, $g^+$)
res$^0$ := MaximumRecur(call1, MaximumRecur(call2, call3))

{ Recursive calls for top coefficient $= +1$ }
call4 := SaturateRecur($f^+$, $g^0$)
call5 := SaturateRecur($f^0$, $g^+$)
res$^+$ := MaximumRecur(call4, call5)

{ Recursive calls for top coefficient $= -1$ }
call6 := SaturateRecur($f^-$, $g^0$)
call7 := SaturateRecur($f^0$, $g^-$)
res$^+$ := MaximumRecur(call6, call7)

{ Combine the cofactors and update the cache }
res := DD_CombineCofactors(top, res$^-$, res$^0$, res$^+$)
DD_CacheInsert(SaturateRecur, $f$, $g$, res)
return res
```

Figure A.3: Pseudocode of one iteration of the saturation procedure in the OhDD implementation.

Function *MaximumRecur*(f, g)
Input: Two OhDD called f and g.
Output: An OhDD that has, at the bottom of each path from the root to the terminal nodes, the maximum terminal found in the same path in f and g.

```
   { Terminal cases }
   if $f = g$ then return $f$ endif
   if $f = +\infty \vee g = -\infty$ then return $f$ endif
   if $f = -\infty \vee g = +\infty$ then return $g$ endif
   if DD_IsConstant($f$) $\wedge$ DD_IsConstant($g$) then
      return DD_Max($f$, $g$)
   endif

   { Lookup the result in the cache }
   res := DD_CacheLookup(MaximumRecur, $f$, $g$)
   if (res $\neq$ null) then return res endif

   { Recursive calls for each cofactor }
   top := DD_TopVariable($f$, $g$)
   $< f^-, f^0, f^+ >$ := DD_GetCofactors(top, $f$)
   $< g^-, g^0, g^+ >$ := DD_GetCofactors(top, $g$)
   res$^-$ := MaximumRecur($f^-$, $g^-$)
   res$^0$ := MaximumRecur($f^0$, $g^0$)
   res$^+$ := MaximumRecur($f^+$, $g^+$)

   { Combine the cofactors and update the cache }
   res := DD_CombineCofactors(top, res$^-$, res$^0$, res$^+$)
   DD_CacheInsert(MaximumRecur, $f$, $g$, res)
   return res
```

Function *Intersection*(f, g)
Input: Two OhDD called f and g.
Output: The intersection of f and g.

```
   res := MaximumRecur($f$, $g$)
   return Saturate(res)
```

Figure A.4: Pseudocode of the intersection procedure in the OhDD implementation