Abstract Interpretation Techniques for the Verification of Concurrent Systems

Robert Clarisó
{rclariso@lsi.upc.es}

Advisor: Jordi Cortadella

Projecte de Tesi

Programa de Doctorat de Software
Departament de Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya

Abstract

Complexity in the design of software and hardware systems has increased exponentially in the last years. Problems like verification and synthesis require more powerful techniques and a greater degree of automation in order to deal with bigger and bigger designs.

This document describes algorithms for the verification and synthesis of concurrent systems that are based on *abstract interpretation.* Abstract interpretation is a general theory for the analysis of the dynamic behavior of systems, which is used in static analysis of programs, code optimization and verification among other areas. The concept behind abstract interpretation is *abstraction*: analysis problems are solved approximately in such a way that termination is ensured, and therefore fully automation can be achieved. There is a trade-off between precision and efficiency: studying a problem at a high level of abstraction is efficient but imprecise, and vice versa.

Two main results are presented. First, an algorithm for the verification of timed systems where the delays of elements are not specified, but left as *symbols.* The algorithm is capable of discovering the necessary constraints on the delays that guarantee the correctness of the system. Several real examples from the domain of asynchronous circuits have been verified with this technique. The second result is an analysis algorithm that permits the full automation of the synthesis of concurrent systems using a technique called Quasi-Static Scheduling.

Acknowledgements

This work has been partially funded by the following entities: the Ministry of Science, Culture and Sports of Spain, with the grant "Beca de Postgrado para la Formación de Profesorado Universitario (FPU)" and the project TIC2001-2476-C03-01, "Modelización, Análisis y Verificación de Sistemas Heterogéneos (MAVERISH)"; Cadence Design Systems, in the project "Design automation for embedded electronic systems"; the Working Group on Asynchronous Circuit Design (ACiD-WG), IST-1999-29119; and Intel Corporation.

Contents

1 Introduction **9**
- 1.1 Motivation . . . 9
- 1.2 Concurrent systems . . . 11
- 1.3 Approaches to verification . . . 12
- 1.4 Abstract interpretation and the goals of the thesis . . . 12
- 1.5 A practical example . . . 13
- 1.6 Organization of the document . . . 14

2 State of the art **15**
- 2.1 Abstract interpretation . . . 15
  - 2.1.1 Introduction . . . 15
  - 2.1.2 Applications . . . 16
  - 2.1.3 Formal definition . . . 17
  - 2.1.4 Abstractions of numeric values . . . 20
- 2.2 Timing analysis . . . 23
  - 2.2.1 Timing Transition Systems . . . 23
  - 2.2.2 Timed Automata . . . 24
  - 2.2.3 Analysis of Timed Systems . . . 25
- 2.3 Synthesis of Embedded Software . . . 27
  - 2.3.1 Introduction . . . 27
  - 2.3.2 Petri Nets . . . 28
  - 2.3.3 Quasi-Static Scheduling algorithm . . . 29
  - 2.3.4 False path problem . . . 32

3 Goals of the thesis **33**

4 Contributions **37**
- 4.1 Summary of the contributions . . . 37
- 4.2 Timing Verification of Concurrent Systems with Symbolic Delays . . . 37
  - 4.2.1 Introduction . . . 37
  - 4.2.2 A motivating example . . . 38
  - 4.2.3 Timing Verification without Symbols . . . 39
  - 4.2.4 Timing analysis algorithm . . . 40
  - 4.2.5 Verification of safety properties using timing analysis . . . 43
  - 4.2.6 Experimental results . . . 46
  - 4.2.7 Conclusions and future work . . . 47
- 4.3 Scheduling of Concurrent Systems . . . 49

4.3.1 Introduction . . . 49
4.3.2 Quasi-Static Scheduling . . . 50
4.3.3 False path detection . . . 53
4.3.4 Conclusions and future work . . . 55
4.4 Publications . . . 55

5 Future work of the thesis **57**
5.1 Remaining tasks . . . 57
5.2 Working plan . . . 58

List of Figures

1.1 Example: D flip-flop 14

2.1 Galois connections 17
2.2 Semantics as a system of equations 18
2.3 Convergence of systems of equations 19
2.4 Comparison of the different abstractions of a set of values. 20
2.5 Operations on convex polyhedra 22
2.6 TTS modelling the controller of the door of a garage. 24
2.7 TA modelling the controller of the door of a garage. 25
2.8 Representations of timed states 26
2.9 Example: Schedulable and non-schedulable Petri nets 31

4.1 Example: a simple timed asynchronous circuit 38
4.2 Algorithm: Abstract interpretation for timing analysis 40
4.3 Algorithm: clock transfer 42
4.4 Example: the clock transfer function 42
4.5 Algorithm: Analysis of safety properties using timing information 44
4.6 Cyclic behavior to illustrate the widening operator. 45
4.7 Example: GasP FIFO controller 46
4.8 Example: Asynchronous pipeline 47
4.9 Example: `nowick` controller 48
4.10 Example: environmental controller 50
4.11 Flow C specification and generated tasks 51
4.12 Comparison of task-based implementation versus process-based implementation. 52
4.13 Example: a false path 53
4.14 Example: automatic false path elimination 54

List of Tables

1.1 Evolution of complexity in software and hardware 10

2.1 Abstractions of numeric values 16
2.2 Brief comparison of numeric abstractions 20

4.1 Experimental results of timing analysis with symbolic delays 48

Chapter 1

Introduction

1.1 Motivation

In the last 20 years, the common trend shared by software and hardware industries has been an exponential increase in the complexity of designs. On the software side, the availability of higher-level languages and new paradigms such as object orientation, the maturity of sofware engineering techniques and the improvement in processor performance, among other factors, have permitted the development of applications with millions of lines of code. On the hardware side, Moore's law predicted in 1965 that the number of transitors integrated in a chip would double every 18 months. Currently, the scale of integration allows the implementation of chips with millions of transistors. Figure 1.1 presents some examples that try to illustrate this tendency. With this increase in complexity, finding powerful techniques to automate the design process has become a major concern in the industry and academia.

Formal verification

Even assuming very low error rates, complex designs will have more bugs, that should be detected before the system is finally deployed. 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. Recent history is full of stories related to the potentially devastating effects of software and hardware errors:

Patriot missile truncation error (1991): [95] The Patriot missile had an internal clock that stored the number of seconds in a floating-point 24 bits register. After long periods of operation, the value in the register lost precision due to the truncation of the floating-point number. As a result, a Patriot missile missed its target, hitting a barracks and killing 28 people.

Pentium FDIV bug (1994): [32] Intel shipped the Pentium processor with a bug in the floating-point division unit. Even though statistically the error only occurred once in 100 milion divisions, this incident caused economic losses to Intel Corporation (500$ milion spent to replace defective chips) and damaged its reputation.

Ariane 5 rocket explosion (1996): [82] A conversion from a 64-bit floating point number to a 16-bit integer caused an overflow in one of the software components controlling the rocket. The result of this error was the explosion of the rocket 37 seconds after the liftoff.

Error detection is typically performed through *testing* and *simulation.* However, the main problem of testing is that it cannot cover all possible situations. Typically, there will be corner cases that will not be

Processor	Year	Transistors
4004	1971	$2,250$
8008	1972	$2,500$
8080	1974	$5,000$
8086	1978	$29,000$
286™	1982	$120,000$
386™	1985	$275,000$
486 DX	1989	$1,180,000$
Pentium ©	1993	$3,100,000$
Pentium II	1997	$7,500,000$
Pentium III	1999	$24,000,000$
Pentium 4	2000	$42,000,000$

Stable Version	Year	Lines of code
Samba 1.9.X	1996	50,000
Samba 1.9.17	1997	60,000
Samba 1.9.18	1998	90,000
Samba 2.0.0	1999	180,000
Samba 2.0.7	2000	200,000
Samba 2.0.8	2001	360,000
Samba 2.2.3	2002	475,000
Samba 2.2.7	2003	600,000

Table 1.1: Left: Evolution in the number of transitors in the processors of the Intel family (Source: `http://www.intel.com/research`). Right: Estimates of the source code sizes of Samba, an Open-Source project that provides file and print services to SMB/CIFS clients and is included in most Linux distributions (Source: `http://statcvs.sourceforge.net`).

covered by the test patterns. Quite often, errors can happen precisely in those corner cases that were not considered during the design. Clearly, there will be 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, and this 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.

Synthesis of concurrent systems

Complex systems like embedded systems are better described at a high level of abstraction. An example of a natural high-level representation is a set of communicating processes that execute concurrently. However, the execution of such a representation will likely occur in a platform with a single sequential processor. Ideally, the high-level description should not impose a performance penalty to the sequential execution. However, a naive implementation will suffer this penalty, mainly due to the time spent by the operating system switching context from one process to the other.

Static scheduling is one of the areas of interest in embedded system design. It comprises a set of techniques that try to generate a sequential implementation, the *schedule*, from a set of concurrent processes, such that the run-time is minimized. Therefore, static scheduling tries to perform as much work as it can at compile-time, reducing the run-time overhead. However, the schedule can not be fully determined statically, because that problem would be undecidable. The challenge lies on performing the most work statically while still remaining decidible and thus *automatic*. Powerful analysis techniques are required in order to perform this task.

1.2 Concurrent systems

The work presented in this thesis will study the verification and synthesis of *concurrent* systems. Verification of concurrent systems requires additional efforts because, for a set of events, all possible interleavings have to be studied. For example, if events (abc) are concurrent, they could happen in many orders such as $(abc - acb - bca - bac - \dots)$.

Some examples of concurrent systems are *asynchronous circuits*, *embedded systems* and *concurrent programs*.

Asynchronous circuits

Asynchronous circuits are circuits where there is no global clock to synchronize its different components. In conventional *synchronous* design, a global clock is used and the result of each component is stored in a latch at each clock cycle. However, clocks consume a lot of power, and it is also difficult to propagate the clock through all the circuit. Also, most of the activity in the circuit occurs close to the clock edge, causing high electromagnetic emissions. Asynchronous circuits solve these problems by replacing the global clock by a local hand-shake between components. This approach grants lower power consumption, less electromagnetic emissions and better efficiency due to early completion, i.e. there is no need to wait for a clock cycle to propagate a result.

The fact that there is no clock makes the verification of asynchronous circuits more complex, because there is more concurrency: events that in a synchronous circuit would happen sequentially can happen concurrently in an asynchronous circuit. Also, the lack of clock makes the circuit more dependent on *timing constraints* that ensure the correcteness of the synchronization within the circuit. This means that the correctness of the circuit depends on the delays of its gates and wires.

Embedded systems

Embedded systems are specialized computation units that perform a set of tasks within a larger system called *environment.* Some examples of embedded systems can be a VCR, a cellular phone, an electronic organizer, or a digital camera. The implementation of these systems is generally a software component running on top of a hardware architecture that might include several CPUs, co-processors, and so on.

Embedded systems can be seen as a set of tasks that are executed *concurrently.* The interaction between this tasks and the environment is typically *reactive*, i.e. the tasks must process inputs from the environment under some timing constraints such as the speed (throughput) or the delay (latency). For these reasons, two problems have special relevance in the area of embedded system design are: *synthesis* of efficient sequential implementations from concurrent specifications; and *verification* of the correctness of the concurrent specification, i.e. ensure that the response of the system is the one expected by the environment and that it satisfies the timing requirements.

Concurrent programs

Concurrent software such as multi-threaded or distributed applications are also hard to analyze due to concurrency. Verification is also more complex because, in general, programs have more data-dependent control-flow choices that hardware specifications.

In these domains, the analysis of data values within the system could provide very useful information. In the case of embedded systems and concurrent software, the result of control-flow choices could be analyzed statically. One could analyze, for example, whether the condition of a `if` statement can be satisfied at run-

time. In the case of asynchronous circuits, accurate timing information could be computed, such as which constraints on the gate and wire delays are sufficient to avoid failures.

1.3 Approaches to verification

Verification is the formal procedure of checking that a system satisfies a specification. The properties that can be described in a specification are typically of two kinds, *safety* properties and *liveness* properties. Safety properties describe conditions that must *always* hold, e.g.

> "The traffic light should not be green for cars and pedestrians simultaneously".

Whenever a safety property is not satisfied, an error occurs. Therefore, safety properties are used to describe something negative that should not happen in the system. On the other side, liveness properties describe conditions that should *eventually* hold, e.g.

> "At some point in the future, the traffic light for pedestrians should become green".

Contrary to safety properties, liveness properties describe something positive that should happen in the system at some point. The fact that a state will eventually be reached (*progress*) and any event of a choice that is repeated infinitely often is eventually taken (*fairness*) fall into this category.

There are several approaches for the automatic verification of these properties. *Model checking* [31] is an automated technique that, given a *finite-state model* of a system and a *property*, checks whether the property holds for a given initial state of the model. *Theorem proving* [23] 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* [41] 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.

1.4 Abstract interpretation and the goals of the thesis

In this thesis, we will study the problems of automatic verification and synthesis for concurrent systems in the presence of data values. Concurrent systems can have clocks, delays, parameters, internal variables or other elements whose value can be relevant to verification and synthesis. For verification, we would like to discover properties like:

> "The system is correct if the value of k is ≥ 3"
>
> "The system is correct if the delay of the sequence of events $a + b + c$ is smaller than the delay of d."

For synthesis, we would like to automatically recognize properties like:

> "Process p will send at most N messages to process q. Thus, messages between p and q can be stored in a bounded queue of size N."
>
> "Event a in process p will never happen, so we don't need to synthesize code for it."

With some exceptions, data analysis is not a first-class citizen of the existing verification and synthesis techniques. Our goal will be the study of efficient algorithms and data structures for the automatic analysis of these data values for the purposes of verification and synthesis. The main concerns of the proposed techniques will be *automation*, *efficiency* and *scalability* to work with real-life examples.

Among the techniques that are available in the domain of analysis, synthesis and verification, abstract interpretation appears as a good candidate for the tasks to be performed in this thesis. Some of the benefits that abstract interpretation provides for our area of interest are:

- *Theory*: Like theorem proving, 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 intepretation are guaranteed to terminate by the underlying theory. Therefore, abstract interpretation analysis can be fully automated.
- *Analysis of data constraints*: Abstract interpretation is well-suited for the discovery of properties on the data-values. 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 Chapter 2).
- *Trade-off between precision and efficiency*: Abstract interpretation provides a generic framework of analysis. Many different abstractions can be used to represent the state 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 abstraction that represents the state of the system efficiently can be "plugged" into the framework of abstract interpretation very easily.

Two results have been obtained in the areas of *timing verification* and *synthesis.* In timing verification, a technique for analyzing systems with unknown delays is proposed. This technique can compute a set of sufficient constraints that guarantee the correctness of a timed system with respect to timed properties. Several experimental results have been studied, from the domain of asynchronous circuits. In the area of synthesis, an analysis technique which can be used in the synthesis of sequential software from concurrent specifications. One of the approaches for the synthesis of these sequential programs is called *Quasi-Static Scheduling* (QSS). The problem of QSS is that whenever it finds a data-dependent choice, it assumes that any result of the choice could be taken at run-time. Therefore, the amount of execution traces that it has to explore increases dramatically. In some cases, the synthesis procedure might fail to terminate within reasonable time. The proposed analysis technique can predict the outcome of data-dependent choices, thus reducing the amount of execution traces to be explored. For example, this can lead to establishing bounds on the lengths of message queues between processes *at compile-time.*

1.5 A practical example

We will present an example of the results that can be achieved using the techniques described in this document. In particular, we will show one example of timing verification of systems with symbolic delays.

Figure 1.1(a) depicts a D flip-flop [86]. Briefly stated, a D flip-flop is a 1-bit register. It stores the data value in signal D whenever there is a rising edge in the clock signal CK. The output Q of the circuit is the value which was stored in the last clock rising edge. We would like to characterize the behavior of this circuit in terms of the internal gate delays. The flip-flop has to be characterized with respect to three parameters (see Figure 1.1(b)):

- *Setup time*, noted as T_{setup}, is the amount of time that D should remain stable before a clock rising edge.
- *Hold time*, noted as T_{hold}, is the amount of time that D should remain stable after a clock rising edge.

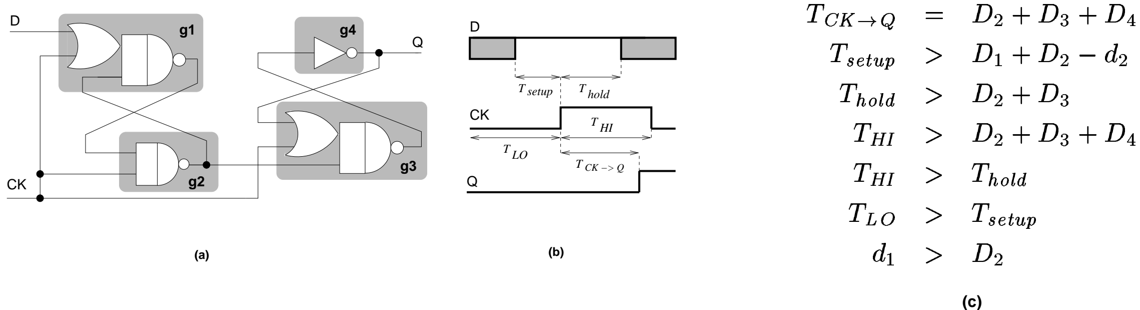


Figure 1.1: (a) Implementation of a D flip-flop [86], (b) description of variables that characterize any D flip-flop and (c) sufficient constraints for correctness for any delay of the gates.

- *Delay* or *clock-to-output time*, noted as $T_{CK \to Q}$, is the amount of time required by the latch to propagate a change in the input D to the output Q.

The timing analysis algorithm is capable of deriving a set of sufficient linear contraints that guarantee the correctness of the circuit's behavior. This behavior will be correct if the output Q matches the value of D in the last clock rising edge. Formally, this property can be stated as:

> *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. 1.1(c) reports the set of sufficient timing constraints derived by the algorithm. Each gate g_i has a symbolic delay in the interval $[d_i, D_i]$. Notice that this result has *characterized* the behavior of the circuit for *any* possible delay of the gates.

1.6 Organization of the document

The remaining of the document is organized as follows:

Chapter 2 presents the state of the art. Known techniques for abstract interpretation, timing verification and static scheduling of embedded software are presented.

Chapter 3 describes in detail the goals to be achieved in the thesis. Several specific problems and areas of research related to verification and synthesis of concurrent systems are identified.

Chapter 4 presents the results that have already been obtained. The main contributions are (1) a timing analysis algorithm for concurrent systems with symbolic delays, which can identify the linear constraints that guarantee the correctness of a timed system, and (2) an automatic algorithm for the elimination of false-paths in static scheduling. Finally, this chapter presents the list of publications of the author related to the work presented in this document.

Chapter 5 lists the future work remaining to achieve the final goals of the thesis, as well as a estimation of the working plan required to complete the thesis.

Chapter 2

State of the art

This chapter aims to provide the foundations of abstract interpretation and the areas where a contribution have been made, *timing verification* and *synthesis of embedded software*. In Section 2.1, the theory of abstract interpretation is presented, with a special emphasis on techniques that allow an efficient analysis of numeric variables. Section 2.2 describes techniques for modelling and verifying timed systems. Timed systems are relevant to abstract interpretation analysis because time can be considered another numeric variable to be studied. Finally, Section 2.3 describes the problem of synthesizing sequential software from concurrent specifications.

2.1 Abstract interpretation

2.1.1 Introduction

Abstract interpretation [41] is a general theory for the static analysis of systems with a dynamic behavior. The main idea behind abstract interpretation is *approximation.* The dynamic behavior of a system can be too complex to be analyzed precisely. Besides, in order to prove a property, approximate information about the dynamic behavior of the system might be sufficient. A classic example is the computation of the sign of a product $c = a \times b$. The sign of c can be computed from the signs of a and b. Knowing the exact values of a and b is not required.

Abstract interpretation can be applied to many kinds of analysis problems in different types of systems. In order to solve a specific problem, the framework of abstract interpretation has to be adapted to:

- *the properties being studied:* We can define a *state* of a system as the set of values that describe the configuration of the system at any given point. The state may contain information which is not necessary to check a given property. Therefore, in our analysis we can work with an *abstraction*, a simplification of the state that ignores the information of the configuration that is not relevant in the specific problem.

- *the semantics of the system:* The behavior of a system can be defined by identifying a set of *locations* where we require information about the state. The relations among the state of the system in these locations establishes a *system of equations.*

The system of equations can be solved iteratively until a fixpoint is reached. The results of this system of equations provide approximate information about the dynamic behavior of the system.

Example	Abstraction	Citation	Properties
Bounds	Signs	[41]	$x \in \{-, 0, +, \pm\}$
	Intervals	[40]	$K_1 \leq x \leq K_2$
	Octagons	[76]	$\pm x \pm y \leq K$
	Convex polyhedra	[50]	$K_1 \cdot x_1 + \ldots + K_n \cdot x_n \leq K_0$
Congruences	Simple cong.	[60]	$x \equiv K_1 \bmod K_2$
	Relational cong.	[61]	$K_1 \cdot x + K_2 \cdot y \equiv K_3 \bmod K_4$
	Trapezohedral cong.	[74]	$K_1 \cdot x + K_2 \cdot y \in [K_3, K_4] \bmod K_4$

Table 2.1: Abstractions of numeric values

2.1.2 Applications

Abstract interpretation has many applications in many different areas. This section will only describe the applications which are close to the area of research in this thesis. The interested reader can find more information about other applications in [38, 39].

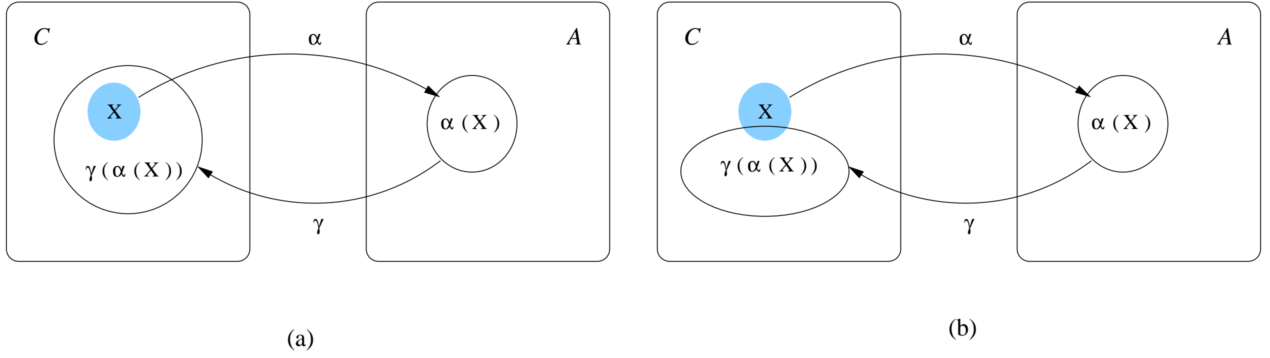
Static analysis is an important area of application of abstract interpretation. It can be applied to the static analysis of *logic programs* [46] and *functional programs* [45, 49]. With these techniques, properties such as *groundness*, *strictness*, *cost estimation* or *termination* can be analyzed approximately. Also, it can be applied to the analysis of *imperative* programs. A problem which has had a lot of success in this area is analyzing the values of program variables. These techniques rely on abstractions that represent numeric values efficiently. In general, these abstractions can be divided into two categories: *bound based* abstractions and *congruence based* abstractions. Figure 2.1 provides a brief summary of these techniques.

Bound abstractions are based on computing bounds on the values of variables. For example, interval analysis computes upper and lower bounds for variables; octagon analysis computes the upper bound of the sum or difference or pairs and variables; and linear relation analysis using convex polyhedra computes the linear invariants satisfied by a set of variables. This information 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”.

Congruence abstractions provide information based on congruence relations. There are several kinds 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, the compiler can use this information for scheduling, loop parallelization or reordering of loops. This results have been further extended to the general problem of *alias analysis*, with specific abstractions that deal with pointers called *shapes* [33, 52, 81]

Several extensions have been proposed to the previous techniques in order to handle recursivity [42], interprocedural analysis [20], floating-point operations [6] and parallel programs [44]. Combining all these techniques, very precise information about the run-time behavior of a program can be collected . This information can be used, for example, to solve the problem of *compile-time detection of run-time errors of imperative programs* [88].

Abstract interpretation has also been applied to the verification of systems other than software. The concept of approximation has been used in [53] to verify real-time systems. Linear relation analysis has been used to verify synchronous programs with counters and hybrid automata [63]. Finally in [19], the



(a)

(b)

Figure 2.1: (a) A Galois connection $\langle \alpha, \gamma \rangle$. (b) is not a Galois connection, because $\gamma(\alpha(X))$ is not an overapproximation of X.

design of a specific abstraction to analyze a concrete real-time system is discussed.

2.1.3 Formal definition

The theory that will be presented in this section has been established in [37, 40–42, 47, 48, 81]. Formally, abstract interpretation can be defined as the computation of an upper approximation of the semantics of a program. The semantics of the program can be expressed as a system of fixpoint equations, which is solved approximately. In order to achieve sound results, we restrict ourselves to approximations fulfilling some desirable properties. Approximations that fulfill these properties are called *Galois connections.*

Galois connection

The states of a program 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 ($\alpha : C \rightarrow A$) and the *concretization* function ($\gamma : A \rightarrow C$) which define the path from concrete to abstract value and vice versa. There are pair of functions should ensure *safety*, i.e. that our abstraction is always an *overapproximation* of the concrete values. This notion can be formalized as a Galois connection.

Definition 1 *Galois connections [48]*
Let $(C, \leq)$ *and* $(A, \sqsubseteq)$ *be partially ordered domains, called the* concrete *domain and the* abstract *domain respectively. A pair of functions* $\alpha : C \rightarrow A$ *and* $\gamma : A \rightarrow C$ *is a* Galois connection *if and only if the following holds:*

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

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

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 most imprecise element of 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 not lose safety in the sense that it will always be an *overapproximation.* This means that the following properties will be satisfied:

$$\forall x \subseteq C : x \subseteq \gamma(\alpha(x))$$
$$\forall y \subseteq A : y \subseteq \alpha(\gamma(y))$$

```
{ X0 }
if ( a >= 0 ) then
  { X1 }
  a := a + 1;
  { X2 }
else
  { X3 }
  a := a - 1;
  { X4 }
endif
{ X5 }
```

(a)

$$
\begin{aligned}
X_0 &= \{\text{Precondition}\} \\
X_1 &= X_0 \cap (a \geq 0) \\
X_2 &= X_1[a := a+1] \\
X_3 &= X_0 \cap (a < 0) \\
X_4 &= X_4[a := a-1] \\
X_5 &= X_2 \cup X_4
\end{aligned}
$$

(b)

$$
\begin{aligned}
X_0 &= \{a = -5 \vee a = 3\} \\
X_1 &= \{a = 3\} \\
X_2 &= \{a = 4\} \\
X_3 &= \{a = -5\} \\
X_4 &= \{a = -6\} \\
X_5 &= \{a = -6 \vee a = 4\}
\end{aligned}
$$

(c)

$$
\begin{aligned}
X_0 &= \{\text{Precondition}\} \\
X_1 &= X_0 \cap [0, +\infty] \\
X_2 &= X_1[a := a+1] \\
X_3 &= X_0 \cap [-\infty, -1] \\
X_4 &= X_4[a := a-1] \\
X_5 &= X_2 \cup X_4
\end{aligned}
$$

(d)

$$
\begin{aligned}
X_0 &= [-5, 3] \\
X_1 &= [0, 3] \\
X_2 &= [0, 4] \\
X_3 &= [-5, -1] \\
X_4 &= [-6, -2] \\
X_5 &= [-6, 4]
\end{aligned}
$$

(e)

Figure 2.2: (a) An imperative program, (b) the exact system of equations considering that the state is only the value of variable i and (c) a forward solution for a given precondition; (d) the approximate system of equations using intervals and (e) a forward solution for the same precondition. and (c) a possible forward.

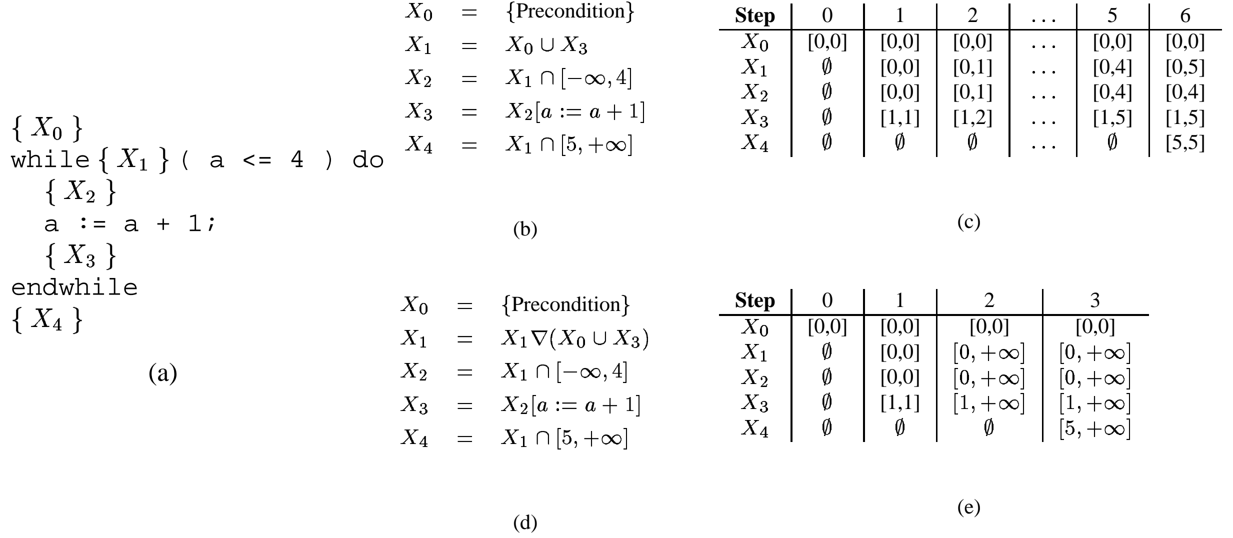
Fixpoint equations

Programs, and systems in general, can be formalized as a system of fixpoint equations that define its behavior. The way in which the system of equations is defined depends on the semantics of the system. In general, we have to define a *state*, the set of values that characterizes the state of the program, and a set of *contexts* where this state is transformed. There is, of course, an initial context with an initial state that is called the *precondition.* Figure 2.2(b) shows a simple example for a small imperative program. The same process could be done for more complex programs [20].

Solving this system of equations gives a description of the semantics of the program. By semantics we understand the set of states that can be achieved in each context starting with a given precondition. This concept is called *forward semantics* [43], and similarly a *backward semantics* can be defined. Figure 2.2(c) shows a forward solution for the system of equations in 2.2(b).

A *forward increasing* [41, 81] solution to a system of fixpoint equations can be obtained through the following procedure: (i) the initial context is initialized with the precondition; (ii) other contexts are initialized with the empty set of states; (iii) equations are applied until convergence is reached, i.e. for a step K, $\forall$ context i : $X_i^K \subseteq X_i^{K+1}$. Conversely, backward and decreasing solutions can be defined [41]. The ideas in this simple algorithm can be extended in order to accelerate the convergence of the system of equations [21, 27, 81].

Finding the exact solution to the system of equations can be very complex. But if a Galois connection is defined, the problem can be translated into solving an approximate system of equations. The solution to this approximate system of equations is a safe overapproximation of the solution, by the definition of Galois connection. Figure 2.2(d) shows the approximate system of equations for the abstract domain of intervals, and 2.2(e) shows the forward solution to this approximate sytem of equations. Notice that the



```
{ X0 }
while { X1 } ( a <= 4 ) do
  { X2 }
  a := a + 1;
  { X3 }
endwhile
{ X4 }
```

(a)

$$X_0 = \{\text{Precondition}\}$$
$$X_1 = X_0 \cup X_3$$
$$X_2 = X_1 \cap [-\infty, 4]$$
$$X_3 = X_2[a := a + 1]$$
$$X_4 = X_1 \cap [5, +\infty]$$

(b)

Step	0	1	2	...	5	6
X_0	[0,0]	[0,0]	[0,0]	...	[0,0]	[0,0]
X_1	$\emptyset$	[0,0]	[0,1]	...	[0,4]	[0,5]
X_2	$\emptyset$	[0,0]	[0,1]	...	[0,4]	[0,4]
X_3	$\emptyset$	[1,1]	[1,2]	...	[1,5]	[1,5]
X_4	$\emptyset$	$\emptyset$	$\emptyset$	...	$\emptyset$	[5,5]

(c)

$$X_0 = \{\text{Precondition}\}$$
$$X_1 = X_1 \nabla (X_0 \cup X_3)$$
$$X_2 = X_1 \cap [-\infty, 4]$$
$$X_3 = X_2[a := a + 1]$$
$$X_4 = X_1 \cap [5, +\infty]$$

(d)

Step	0	1	2	3
X_0	[0,0]	[0,0]	[0,0]	[0,0]
X_1	$\emptyset$	[0,0]	$[0, +\infty]$	$[0, +\infty]$
X_2	$\emptyset$	[0,0]	$[0, +\infty]$	$[0, +\infty]$
X_3	$\emptyset$	[1,1]	$[1, +\infty]$	$[1, +\infty]$
X_4	$\emptyset$	$\emptyset$	$\emptyset$	$[5, +\infty]$

(e)

Figure 2.3: (a) An imperative program with a loop, (b) the system of equations for intervals without widening; (c) the number of steps required to reach a solution without widening; (d) the system of equations for intervals with widening and (e) the solution to the system of equations with widening.

solution to the approximate system of equations ($a \in [-6, 4]$) is an overapproximation of the real solution ($a = 4 \vee a = -6$).

The system of equations in Figure 2.2 did not contain cyclic references in equations, but in general, systems of equations may contain cycles. For example, in order to define the system of equations of a loop, we require cycles. When cycles are introduced, convergence of the solution is no longer guaranteed. Even if the system of equations converges, it may require a very high number of steps to converge, making it impractical for analysis. Figure 2.3 shows an example of a program, whose system of equations in (b) contains cycles. Solving this system of equations requires a number of steps which depends on the number of iterations of the loop at run-time. Obviously, this is not practical for a static analysis.

Again, abstract interpretation solves this problem using approximation. A special operator called *widening* (∇) is defined. This operator is used in the equation that define cyclic relations and guarantees convergence by definition. Intuitively, it works as an induction step, assuming that the behavior observed after the loop could be repeated indefinitely.

Definition 2 *Widening operator [48]*

A widening operator ∇ *is a function* $\nabla : C \times C \to C$ *such that*

1. $\forall x, y \in C : x \cup y \sqsubseteq x \nabla y$

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. Figure 2.3(d) shows the system of equations using widening. The widening operator that is being used is

Concept	Intervals	Octagons	Polyhedra
Relational properties	$\downarrow\downarrow$	$\uparrow$	$\uparrow$
Precision of results	$\rightarrow$	$\uparrow$	$\uparrow\uparrow$
Supported operations	$\uparrow\uparrow$	$\downarrow$	$\uparrow$
Complexity	$\uparrow\uparrow$	$\uparrow$	$\downarrow\downarrow$
Tool support	$\uparrow$	$\rightarrow$	$\uparrow$

Table 2.2: Brief comparison of numeric abstractions

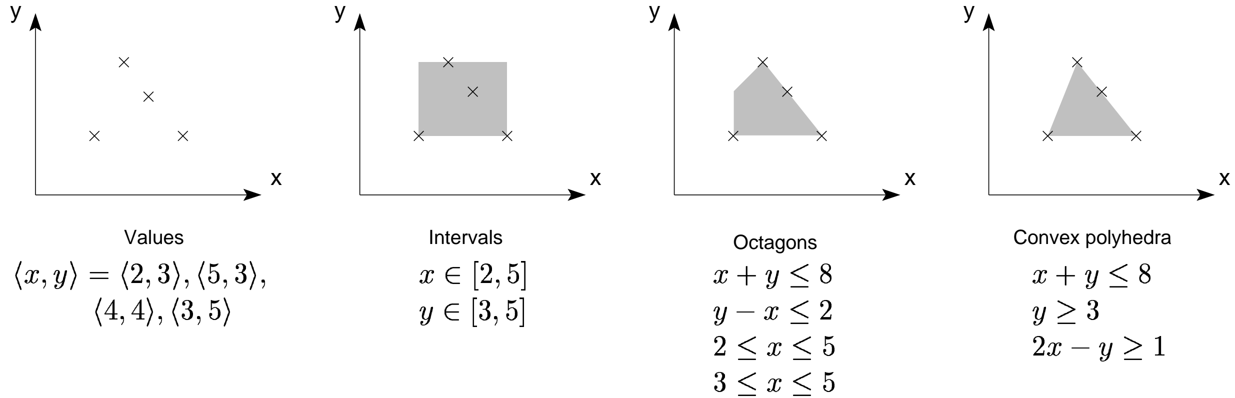


Figure 2.4: Comparison of the different abstractions of a set of values.

the standard widening for intervals:

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

From the definition, we can see that this operator can be applied at most twice before reaching convergence: any change in the lower or upper bound will send them to infinity, including any further change in the same direction. Using this widening, convergence can be reached in fewer steps, as seen in Fig. 2.3(e). However, it should be noted that the widening sacrifices precision in order to ensure convergence. Notice that the solution in 2.3(e) is less precise than that of 2.3(c). The problem of finding efficient yet precise widening operators is very important in abstract intepretation. An example of techniques that can be used to obtain more precise widening operators can be found in [12].

2.1.4 Abstractions of numeric values

The analysis of properties of data values with Abtract Intepretation requires efficient data structures that can represent and manipulate constraints on numeric values. The problem of finding new representations of this kind is an interesting open problem, which is relevant to abstract interpretation and other areas like linear programming or computational geometry.

In this section, we will focus on the abstractions based on *bounds* (see Figure 2.1), because they have had a greater success in the area of verification. Table 2.2 presents a brief comparison between these abstractions, *intervals*, *octagons* and *polyhedra*. Intervals are very simple and efficient, but they cannot represent *relational* constraints, i.e. symbolic constraints between variables like $x = y$. Octagons provide more precision, being able to represent constraints like $x = y + 1$ even if we don't know the variable for x and y.

Finally, convex polyhedra provide the most precise approximation, but most operations with polyhedra have a high complexity.

Figure 2.4 shows an example of a set of values of two variables, x and y, that has been represented using these abstractions. Notice that each abstraction provides an overapproximation of the set of values, and the degree of precision of the approximation varies from one abstraction to the other. Also, notice that all these abstractions represent *convex* sets of values. Whenever a non-convex set of values has to be represented, there is a loss of precision.

Intervals

Intervals are a representation for constraints on the upper or lower bound of a single variable, e.g. $(c_1 \leq x \leq c_2)$. Interval analysis is very popular due to its simplicity and efficiency: an interval abstraction for n variables requires $O(n)$ space, and all operations require $O(n)$ time.

Another strength of interval analysis is the possibility to find precise overapproximations of complex operations such as divisions, products, modulos, ... For example,

$$\begin{aligned} [2,3] * [5,11] &\subseteq [10,33] \\ [3,10]\ /\ [7,14] &\subseteq [0,1] \\ [1,2] \bmod [3,4] &\subseteq [1,2] \end{aligned}$$

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 don't 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.

Octagons

Octagons [76] are an efficient representation for a system of inequalites on the sum or difference of pairs of variables, e.g. $(\pm x \pm y \leq c)$ and $(x \leq c)$. The name is chosen because in a system with two variables, octagons can represent at most eight constraints. One of the advantages of octagons is that they can represent *symbolic* constraints among variables, e.g. $(x = y + 1)$, even if the value of those variables is unknown. Efficiency is another advantage of this representation: the spatial cost for representing constraints on n variables is $O(n^2)$, while the temporal cost is between $O(n^2)$ and $O(n^3)$ depending on the operation.

The problem of this representation is that it loses precision whenever it has to analyze the outcome of assignments like $x := K * y$ or $x := y \pm z$. This problem is inherent to the family of constraints that can be represented with this abstraction.

There is a publicly available implementation of the octagon abstraction, the oct library [75].

Convex Polyhedra

Convex polyhedra [50, 63] are an efficient representation for sets of linear inequality constraints, e.g $(3x + 2y - z \leq 7)$. This abstraction 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 $B \in \mathbb{Q}^m$. 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,

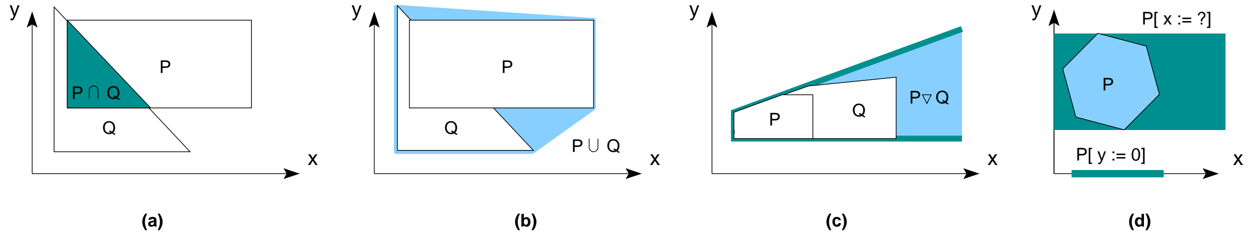


Figure 2.5: Several operations on convex polyhedra: (a) intersection of polyhedra , (b) union of polyhedra as the convex hull, (c) widening of polyhedra and (d) assignment of a linear expression or an undefined value.

because there are efficient algorithms [28,50] that translate one representation to the other, and several of the operations for convex polyhedra are computed very efficiently when the proper representation of polyhedra is available.

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** ($P \bigcup Q$)): 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. $P \bigcup Q$ is defined as the polyhedron with a system of generators that is the union of those in P and Q.

- **Intersection** ($P \bigcap Q$): The intersection of two convex polyhedra is necessarily convex. $P \bigcap Q$ 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. Widening operator must ensure that it will reach fixpoint after a finite number of iterations. $P \nabla Q$ is 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.

- **Applying a linear assignment** ($P[d := Cx + D]$): Linear assignments to a dimension of the polyhedron transform the vertices and the edges of the polyhedron as $V' = \{Cv + D|v \in V\}$ and $R' = \{Cr|r \in R\}$.

- **Assigning an undefined value to a dimension** or **quantifier elimination**($P[d :=?]$): this operation removes all constraints for a given dimension of the polyhedron, while keeping all the implicit constraints about the rest of dimensions intact. This operation is implemented with the Fourier-Motzkin elimination [51] 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 d so the coefficient in d 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 2.5 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 loose precision. All other operators are exact.

There are many libraries of convex polyhedra in the public domain, among others New Polka [80], polymake [59], Qhull [89], PPL (Parma Polyhedra Library) [13] and Polylib [87]. Additional resources on polyhedral computation can be found in [58].

2.2 Timing analysis

There are many formal notations used to model the behavior of real-time systems: Timed Graphs, Timed Transition Systems [64], Timed Automata [5], Hybrid Automata [4] and Timed Petri Nets among many others. In the following, we will present the syntax and semantics of some of these notations, as well as discussing the complexity of its verification.

2.2.1 Timing Transition Systems

Timed transitions systems (TTS) were introduced in [64] 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 [10].

In a transition system, there is a set of *states*, and in each state, there can be several enabled *events*. Whenever an event is fired, the state is changed according to a *transition relation.* The event to be fired is chosen non-deterministically among the enabled events. A sequence of firings of events is called a *run*, and it is a valid run only if at each step the event being fired was enabled in that state.

Definition 3 *Transition system [10]*

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* $s \stackrel{e}{\to} s'$*. An event* e *is enabled at state* s *if* $\exists s \stackrel{e}{\to} s' \in T$*. We will denote the set of events enabled at state* s *by* $\mathcal{E}(s)$*.*

Definition 4 *Run of a TS*

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

TTS introduce the notion of *lower* and *upper delay bounds* of 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.

Definition 5 *Timed transition system [64]*

A timed transition system *(TTS) is a triple* $A = \langle A^-, d, D \rangle$ *where* $A^- = \langle S, \Sigma, T, s_{in} \rangle$ *is a TS called the* underlying transition system*,* $d : \Sigma \to \mathbb{R}^+$ *and* $D : \Sigma \to \mathbb{R}^+ \bigcup \{\infty\}$ *respectively associate a* minimal *and a* maximal delay bounds *to each event, such that* $\forall e \in \Sigma : d_e \leq D_e$*.*

In this model, choosing which event is fired requires checking that the delay bounds of all enabled events are satisfied by our choice. For example, if three events e_1, e_2 and e_3 have become enabled simultaneously, and their delays are $e_1 = [2, 3]$, $e_2 = [4, 5]$ and $e_3 = [1, 7]$, then only e_1 or e_3 can be chosen to fire. We cannot choose to fire e_2, because it cannot fire before 4 time units and in that time e_1 would be forced to fire by its upper delay bound. Therefore, the definiton of a valid run of a TTS has to establish that moment in time where each event is fired is consistent with the delay bounds of the events involved.

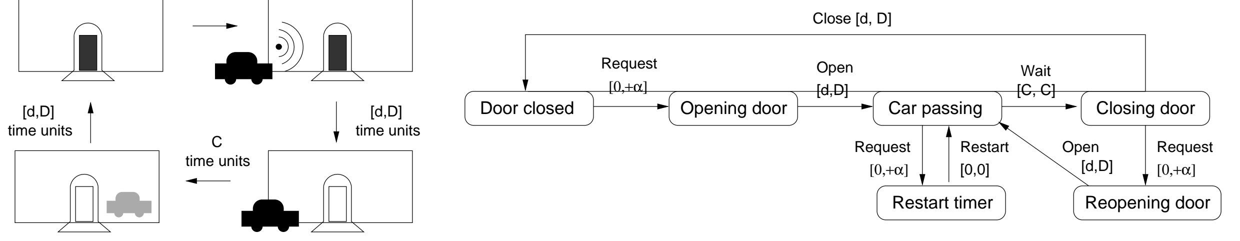


Figure 2.6: TTS modelling the controller of the door of a garage.

Definition 6 *Timed state sequence [64]*

[64] 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 7 *Run of a TTS [64]*

Let $A = \langle A^-, d, D \rangle$ be a TTS. 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:*

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

Example. Figure 2.6 shows a TTS modelling the controller of the door of a garage. There are several cars in the garage, and each car owner has a remote control that can be used to request the door to be opened. After receiving a request, the door opens and remains open for C time units. If there is any request during this time, the timer is restarted, otherwise, the door closes automatically. The time required by the door to become opened or closed is between d and D time units.

2.2.2 Timed Automata

Timed automata (TA) were introduced in [5] as a formal notation to model the behavior of real-time systems. A TA is an automaton extended with a set of *clock variables* and *clock constraints*.

Definition 8 *Clock constraints [3]*

For a set of clocks X, the set of constraints *$\Phi(X)$ is defined by the grammar:*

$$\Phi(X) := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \phi_1 \wedge \phi_2$$

where x is a clock in X, c is a constant in $\mathbb{Q}$ and ϕ_1 and ϕ_2 are constraints in $\Phi(X)$.

Each state, called *location*, has an associated clock constraint called *invariant* that describes the valid values that clocks may have in that location. Transitions between states are called *switches*, and they are extended with a clock constraint called *guard*. A switch can only occur if the guard is satisfied by the values of clocks. After the switch, some of the clock variables can be reset to zero.

Definition 9 *Timed Automata [3]*

A timed automaton *is a tuple $T = \langle L, L_{in}, \Sigma, X, I, E \rangle$ where L is a finite set of* locations*, $L_{in} \subseteq L$ is a set of initial locations, Σ is a finite set of* labels*, X is a finite set of* clocks*, I is a mapping called* invariant *that labels each location $s \in L$ with a clock constraint in $\Phi(X)$, and $E \subseteq L \times \Sigma \times 2^X \times \Phi(X) \times L$ is the*

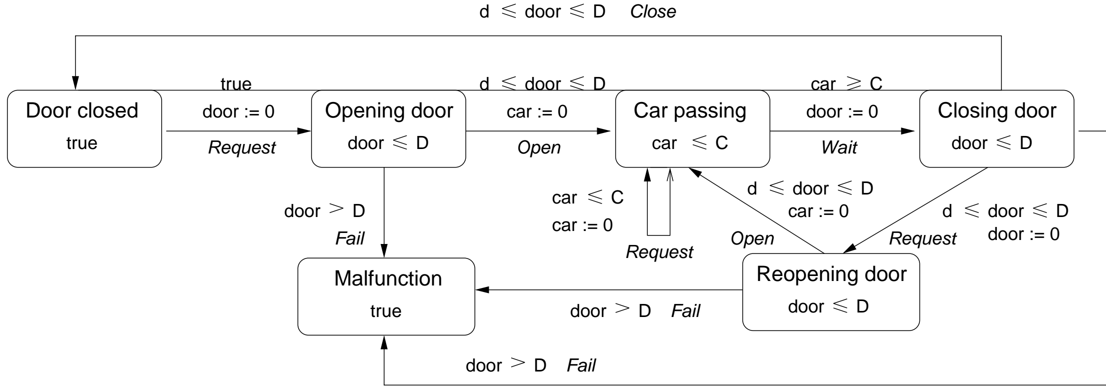


Figure 2.7: TA modelling the controller of the door of a garage.

set of switches*. A switch $\langle s, \sigma, \varphi, \lambda, s' \rangle$ represents a transition from location s to location s' on a symbol σ. φ is a clock constraint over X called* guard *that specifies when the switch is enabled, and $\lambda \subseteq X$ gives the set of clocks to be reset to zero with this switch.*

Contrary to TTS, in TA there are two ways in which the state of the system might evolve: *time passing* or *execution of switches*. In time passing, time can elapse while the automaton stays in a specific location. In this scenario, the invariant of the location has to be preserved, i.e. after the time elapsed, the invariant should still hold. The other way of changing the state is a switch from one location to another. In this case, the guard of the switch should be satisfied, and after resetting the clocks of the switch to zero, the invariant of the new location should also be satisfied.

Definition 10 *Semantics of a TA [3]*
The state of a timed automaton is defined as a pair $\langle s, \nu \rangle$ where s is a location and ν is a clock valuation that satisfies $I_s(\nu)$. Such a state can be changed according to one of the following transitions:

- *Time passing: For a time increment $\delta \geq 0$, the state can evolve as $\langle s, \nu \rangle \xrightarrow{\delta} \langle s, \nu + \delta \rangle$ if $\forall d : 0 \leq d \leq \delta : I_s(\nu + d)$ is satisfied.*
- *Execution of a switch: For a switch t of the form $\langle s, \sigma, \varphi, \lambda, s' \rangle$, the state can evolve as $\langle s, \nu \rangle \xrightarrow{t} \langle s', \nu[\lambda := 0] \rangle$ if ν satisfies the guard φ and $I_{s'}(\nu[\lambda := 0])$ is satisfied.*

Example. We will show an example of TA modelling an extension of the example used for TTS. In this case, the mechanism that opens and closes the door of the garage automatically can malfunction. This situation can be detected if the door of the garage requires more than D time units to become opened/closed. In that case, the door will ring a bell as a warning. This extended system has been modeled in Figure 2.7. Contrary to TTS, there are explicit clock variables, called *car* and *door*. These clocks have to be reset explicitly in transitions, but time passing is implicit within states.

2.2.3 Analysis of Timed Systems

Verification of concurrent systems has achieved very successful results. This success can be related to efficient techniques to traverse or generate the reachable state space. For example, *Symbolic techniques*, based on Binary Decision Diagrams, can represent large state spaces symbolically. There are techniques that operate only on symbolic representations. Such fully symbolic techniques can handle state spaces with

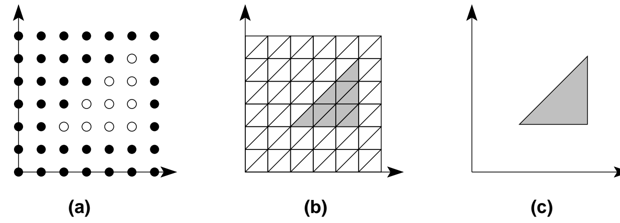


Figure 2.8: Representations of timed states: (a) discrete-based representation, (b) region-based representation and (c) zone-based representation.

sizes that can not be represented explicitly. *Bitstate hashing* can be used to explore only a subset of the state space. *Partial order reductions* decrease the number of interleavings of concurrent events that have to be explored.

Several of these techniques can also be applied to the verification of timed systems. However, the complexity of timed verification is much higher than that of untimed verification. For each untimed state, we have to consider to possible moments in time where it can be entered or exited. This aggravates even more the state explosion problem. Efficient data structures and algorithms are required to represent *collections* of timed states.

Timing analysis with constant delays

Many results have been achieved in the verification of hybrid systems and timed automata. In hybrid automata, state spaces are analyzed using convex polyhedra [63]. In timed automata, only a restricted class of clock constraints is supported (see definition 8). Thanks to this restriction, the timed state space of a timed automaton can be represented efficiently. There are three major families of representations, which can be seen in Figure 2.8:

- *Discrete representations* [25]: Clocks are assumed to take only natural valuations $t = 0, 1, 2, \ldots$. In some models, this might fail to capture the behavior of the model. For example, timed automatas with constraints like $(clk < c)$ will not be analyzable exactly using discrete time.
- *Region-based representations* [2]: A region is a set of clock valuations such that (i) the integer part of all clocks are equal and (ii) the order between the fractional part of timers is the same.
- *Zone-based representations* [54]: A zone is a convex set of regions. This set can be represented very efficiently as constraints in the bounds of clocks $(c_1 \leq clk \leq c_2)$ and differences between pairs of clocks $(clk_1 - clk_2 \leq c)$. There is an efficient implementation of this set, which is called *Difference Bound Matrix* (DBM).

In the case of discrete and region time representations, the complexity lies in the representation of *sets* of clock valuations. Zone-based representations already describe set of clock valuations, but they are limited to *convex* sets, while clock valuations may be *non-convex*. In order to keep track of possible non-convex sets of clock values, a list of zones should be kept for each untimed state. Storing and manipulating these sets of zones/regions/discrete values is the bottleneck of timing analysis techniques. Several approaches can be used to operate efficiently with these sets. For instance, *Difference Decision Diagrams* (DDD) [77] is a symbolic representation for non-convex sets of zones which shares properties from DBMs and BDDs. A DDD is like a BDD where each node, instead of being a boolean variable, is condition of the form $(x - y \leq c)$. Another representation that is based on DBMs is *Numeric Decision Diagrams* (NDD) [55], which is a symbolic representation for discrete time. *Region Encoding Diagrams* (RED) [97] is a symbolic

representation for non-convex sets of regions. This representation is also based on BDD-principles. Another symbolic representation for non-convex sets of regions that is based on BDDs is *Clock Difference Diagrams* (CDD) [16]. Timed polyhedra [22] is a representation for non-convex sets of regions in which a non-convex set is defined by the *extremal* regions of the set.

Many of these techniques have been implemented into powerful tools for the analysis of real-time systems. Some examples of these tools are Uppal, Kronos, CMC, Cospan and HyTech.

Timing analysis with symbolic delays

The previous techniques were based on the assumption that the delays of all events of the system are known constants. A more complex problem is the verification of timed systems where delays are not fixed a priori. These unknown delays are represented with symbols, so the problem can be stated as "timing verification of systems with symbolic delays" or "parametric timing verification". For these problems, the previous data structures cannot be used. The lack of an efficient symbolic representation for non-convex sets makes this problem very difficult to deal with. However, several formalisms have been used with moderate success.

- *Presburger arithmetics* is the first-order theory of natural numbers with addition. Even though the verification of a Presburger formula has a polynomial average case complexity, the worst-case complexity is $O(2^{2^{2^n}})$ [83]. This complexity restricts the number of symbols that can be used in a formula. In [7], Presburger arithmetics is used for the verification of timing diagrams. In [8], the problem of *symbolic time separation* of events is studied: for a given pair of events, upper and lower bounds on the separation between them are determined automatically. However, in both cases the size of the examples that can be verified and the number of symbols that can be used is limited by the exponential worst-case behavior.

- *Parametric Difference Bound Matrices* (PDBMs) [9] are an extension of DBMs which supports parameters (symbols). This representation has been used to in the verification of timed automata with symbols [67]. Again, this technique has strong restrictions on the size of the system and the number of symbols due to the high complexity of the approach.

There are many open problems in this area. From the theoretic point of view, decidibility and complexity of several problems should be established. From an applied point of view, efficient data structures and algorithms should be designed and evaluated, focusing on the verification of larger systems. Our contributions in this area can be found in Section 4.2.

2.3 Synthesis of Embedded Software

2.3.1 Introduction

This section will be focused on the problem of *static scheduling of concurrent systems*. In this problem, the input is a concurrent specification in some formal model that allows the representation of control and data [24, 56, 91]. The output is a sequential implementation that can be executed on a single processor without operating system support. Notice that the goal of this problem is different to that of *static scheduling of real-time systems*, where the aim is defining an ordering of timed tasks that satisfies a set of deadlines.

For systems with data-dependent choices, the static scheduling problem is undecidable in general [24]. In these systems, the scheduling problem is relaxed to *quasi-static scheduling*, where most of the scheduling is done at compile-time *but* the data-dependent choices are postponed until run-time.

The problem of static scheduling of concurrent systems is deeply related to the kind of representation that is being used to model the embedded system. There are many models for the representation of embedded systems. Data flow networks [72], Communicating Sequential Processes [66] and Kahn process networks [69] are classic examples of concurrent representations. The static scheduling problem for these representations has been studied in [24, 73].

Esterel [18] is a concurrent language that describes systems synchronized to a global clock. *Lustre* [62] is a synchronous dataflow language, that defines the system behavior as a set of definitions that are evaluated at every clock cycle. A survey on techniques for the synthesis of Esterel and Lustre can be found in [57]. The research project Polis [14] describes systems using *Co-design Finite State Machines (CFSM)*, an extended asynchronous version of Finite State Machines (FSM) which describe both data and control. The static scheduling inside Polis is described in [29]. Its successor, the research project Metropolis, uses a high-level representation called *Metropolis Meta-Model* [15], which defines a CSP-like system with communication and synchronization primitives. This representation provides a meta-model in the sense that this formalism is general enough to represent several models of computation. The same genericity is possessed by *Funstate* [92], in which data and control flow are kept separately.

Petri Net based models are very popular because of the underlying theory and an intuitive graphical notation. The notion of Petri Net defines the control of a system in a way that exposes parallelism explicitly. However, data is not incorporated into the basic notion, and several extensions of Petri Nets have been proposed in order to define a modelling language for embedded systems. Most of these extensions belong to the general family of *coloured Petri Nets* [68], but some of these representations are worth mentioning because they are specially devised for embedded systems. *Extended Time Petri Nets (ETPN)* [85] extend the Petri Net models with a separate directed graph modeling the datapath. *Petri Net based Representation for Embedded Systems (PRES)* [36] merges data into the Petri Net, where places represent variables and tokens represent values; there are additional guards on the values that should be satisfied before firing a transition. *Dual Transition Petri Nets (DTPN)* [96] also present data and control in the same Petri net model, this time by distinguishing two kinds of transitions: control related and data-related. *FlowC* [35] is a representation based on Petri Nets where each transition is labelled with C code that should be executed atomically if the transition is fired.

This last technique is specially relevant to the area of this document, because it exhibits a problem called *false path problem* which can be addressed using abstract interpretation. The rest of this section describes in some detail this Quasi-Static Scheduling (QSS) algorithm which works on a Petri Net representation of processes. In section 4.3, the contribution to solve the *false path problem* is presented..

2.3.2 Petri Nets

This section describes general notions of Petri Nets used in the description of the algorithm. Further theory, definitions and properties of Petri Nets can be found in [79].

A Petri Net is a 4-tuple $PN = (P, T, F, M_0)$ where $P = p_1, \dots, p_n$ is a finite set of *places*, $T = t_1, \dots, t_m$ is a finite set of *transitions*, $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the *flow relation* and $M_0 : P \to \mathbb{N}$ is the *initial marking*. This tuple 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. Transitions are tipically drawn as bars or boxes, while places are represented by circles.

Each place on the Petri Net contains a number of *tokens*, defined by a *marking* $M : P \to \mathbb{N}$. The number of tokens in a place p at a marking M is denoted by $M[p]$. The initial marking describes the initial state of the places of the net. A marking might enable a transition, meaning that this transition can be fired; firing an enabled transition modifies the marking, possibly enabling or disabling other transitions. Formally, 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 $M'[p] = M[p] - F(p,t) + F(t,p)$ for each $p \in P$. In the sequel, $M[t\rangle M'$ denotes the fact that marking M' is reached from marking M by firing the enabled transition t.

A transition is called a *source* transition when $F(p,t) = 0$ for all $p \in P$. Source transitions are always enabled independently of the current marking, and therefore they can always be fired at any time. By this reason, these transitions will be used to model input events from the environment.

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 set of reachable markings from the initial marking is denoted by $[M_0\rangle$. The *reachability tree* of a Petri Net is a tree that contains nodes labeled with the marking in $[M_0\rangle$; the root node is labeled with M_0 and a node labeled with M has a child M' if there is a transition t such that $M[t\rangle M'$. Each path starting at the root of the reachability tree represents a sequence of transitions fireable from M_0.

A key notion we use in Petri Nets to define schedules is *equal conflict sets*. A pair of transitions t_i and t_j is said to be in *equal conflict* if $F(p,t_i) = F(p,t_j)$ for all $p \in P$. These transitions are in conflict in the sense that t_i is enabled at a given marking only if t_j is enabled, i.e. if the firing of a transition disables t_i it also disables t_j. The equal conflict is an equivalence relation on the set of transitions, that can be partitioned in set of equivalent transitions called *equal conflict sets* (ECS). For example, the set of source transitions is an ECS, which is denoted by E_U. By definition, if one transition of an ECS is enabled at a given marking, all the other transitions of the ECS are also enabled. Thus, we may say that this ECS is enabled at the marking.

A place p is said to be a *choice* place if its has more than one successor transition. A choice place is *Equal Choice* if all the successor transitions are in the same ECS.

2.3.3 Quasi-Static Scheduling algorithm

Assumptions

The QSS algorithm presented on this section works under a set of assumptions on the environment and the system. These assumptions must hold in order for the QSS algorithm to be appliable.

We consider a system to be specified as a set of concurrent processes. The implementation of this system is mapped as software to be executed on a single programmable processor.

The system interacts with the environment through *input* and *output* ports. The interaction proceeds as follows: the environment can place an object in an input port at any time; the system reacts to this input by performing some actions, which may result in writing an object to an output port; the environment can consume the objects in output ports at any time.

The only assumption made on the inputs is about the *rate*: the rate of the arrivals of inputs cannot be faster than the rate of processing of inputs. This assumption is necessary because otherwise it would be impossible to avoid overflow. No other assumptions are made on the inputs, in particular we assume that there is no correlation among inputs.

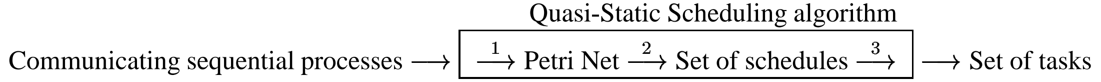
The behavior of each process in the system is described by a sequential program, written in FlowC. Communication among processes is point-to-point and occurs through uni-directional FIFO ports.

Overview of the QSS algorithm

Under these assumptions, the QSS algorithm will generate a sequential task for each input port. The tasks are guaranteed to require a finite memory for the communication ports, as the QSS algorithm computes a bound to the maximum number of elements in each port of the system. If the algorithm fails to establish bound for any port, QSS fails and the system is considered to be *non-schedulable.* The synthesis of tasks is performed using Petri Nets as the underlying model, due to their explicit representation of concurrency.

The code of each task might contain data-dependent control constructs, like conditionals or loops, because they cannot be resolved without the values of data, that are only available at run-time. All other operations, which are not data-dependent, are sequencialized to reduce the run-time overhead.

Before introducing the QSS algorithm in detail, we will present a very brief overview of the three main steps of the algorithm:



1. *Translation into Petri Net*: The concurrent specification is translated into a single Petri Net model. FlowC statements and data-dependent constructs appear as labels in this Petri Net. Inputs from the environment are modeled as source transitions in the Petri Net.
2. *Scheduling*: The Petri Net is analyzed to find all the possible execution flows that can be taken when an input reaches the system. Those execution flows contain actions that belong to different processes and can be sequentialized. The result is an *schedule* for each input of the system.
3. *Code generation*: The last step consists in generating code for the schedules. The goal of this step is to reduce the size of generated code and to take advantage of bounds in communication buffers to increase performance, e.g. replacing buffers of size 1 by scalar variables.

Translation into Petri Net

The first part of the QSS algorithm is the translation of a set of concurrent processes into a single Petri Net. This translation is achieved in two steps: *compilation* and *linking*. Briefly, compilation generates a subnet for each process in the network, while linking combines those subnets into the final net.

Compilation performs a syntax-driven translation on the code of each process to get its subnet. A place is created for each port read or written from the process. The FlowC statements are translated differently depending on t: statements that do not operate with ports are labeled directly into a Petri Net, while statements that operate with ports are decomposed into several transitions until basic communication statements are reached. These basic communication statements (`SELECT`, `READ`, `WRITE`) are modeled as adding/removing tokens to/from a place related to a port.

When translating a compound statement (e.g. `while` loop) that contains a communication construct, data-dependent constructs are modeled as *Equal Choice* places, labeled with a boolean condition and with two outgoing arcs, labeled `True` and `False`. The succesor transitions of such a place constitute an ECS.

Petri net linking combines all the subnets from all the processes into a single subnet. This is done by (1) merging places that represent the same port into a single place and (2) adding one source (sink) transition connected to each input (output) port. This resulting Petri Net models the concurrency in the execution of the processes of the system. As stated previously, source transitions model inputs from the environment. Source transitions are always enabled, modeling that input transitions can be fired from the environment at any time. However, we assume that the input rate is low enough to allow the processing of inputs in the system.

Scheduling

When the Petri net model for the whole system is available, the next step is the generation of a set of traces that represent all possible behaviors at run-time. The evolution of a Petri Net can be modeled by means of the reachability tree, that specifies which markings are reachable from the initial marking (and hence the possible states of the communication channels), and which transitions can be fired (and hence, which

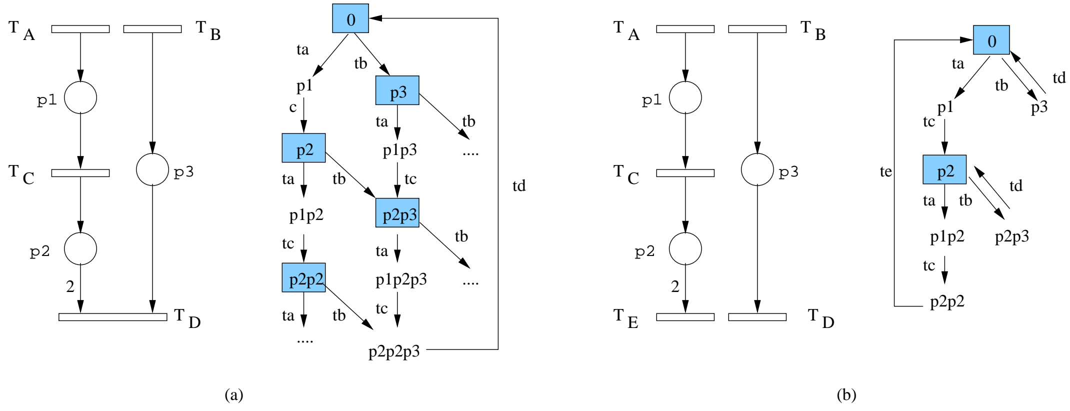


Figure 2.9: (a) Non-schedulable net, (b) Schedulable net with schedule

code will be executed at each point). We do not need to explore the entire reachability space to model the behavior of the system, because we know that this behavior will be cyclic: *wait for input*, *receive input*, *react to input*, *wait for input again*. Therefore, the goal of this phase will be finding a relevant subset of the reachability space, i.e. a relevant subtree of the reachability tree, that is enough to model the cyclic behavior of the system regardless of the values of data, while ensuring finite memory for communication channels. This relevant subtree is called a *schedule*, and it provides a sequential summary of the response of all processes in the system to inputs from the environment. This schedule will be our first step towars a task-based implementation. From this schedule, we will describe the behavior of the system with respect to a single input, and from this model we will generate a task for each input of the system.

An important notion to be defined before describing schedules is the notion of *await state*. A state of the reachability tree where the only transitions that are enabled are the source transitions E_U is called an *await state*. As the only transitions that can be fired are inputs from the environment, an await state models a situation where the system is waiting for the environment to produce an input. Any run of a task of the system will start in an await state (the system was waiting and the environment produces an input) and will end in an await state (the input has been fully processed or it the system needs more inputs to continue its computation).

The definition of schedule is based on this notion of await state. A *schedule* of a PN is finite directed graph, where each node v is associated with a marking $M(v)$ and each edge (u, v) is associated with a transition $T(u, v)$. The graph has 5 properties:

1. There is only one node v associated with the initial marking M_0.
2. For each node v, the out edges are associated with transitions of an ECS that is enabled in marking $M(v)$. When this ECS is E_U, the source transitions, this is an await node. This ECS can map a data-dependent construct from the original specification.
3. For each edge (u, v), marking $M(v)$ can be reached from marking $M(u)$ firing transition $M(u, v)$.
4. Each node has at least one path to an await state.
5. Each await state is at least on one cycle

For example, Figure 2.9 presents the schedulability problem for two Petri Nets. For each PN, we try to build a schedule that ensures the five properties stated above. Await states in the schedules are displayed inside a grayed box. For net (a) we cannot find a finite schedule, thus the net is labeled as non-schedulable.

The impossibility to schedule this net comes from the lack of control of the inputs from the environment `a` and `b`: one can be fired from the environment indefinitely, therefore we can reach a marking with an infinite number of tokens in the places `p2` or `p3`. However, we can find a finite schedule for net (b), as the reaction to one input is independent from the other inputs from the environment.

A complete definition of the scheduling algorithm can be found in [35]. Intuitively, schedules are built as subtrees of the reachability tree. The traversal starts at the initial marking, and it explores the tree looking for a path to an await node. From this await node, the tree is explored further to find paths to other await nodes, and cycles that contain the await node. If the tree built in this way satisfies the 5 properties of a schedule, then a schedule is built by creating the cycles in our subtree. Otherwise, the algorithm declares the PN as non-schedulable and terminates.

If a finite schedule can be found for a given PN, its properties ensure:

- *Cyclic behavior*: Is guaranteed by properties (4) and (5), because an await state can always be reached from the current state, and an await state is always on a cycle.
- *Correct behavior regardless of the values of data*: Data-dependent constructs are not evaluated during scheduling, but at run-time. Therefore, the schedule describes the behavior of the system for any run-time values of data.
- *Finite memory for communication channels*: The schedule specifies a finite set of reachable markings. Each marking describes the number of tokens for each place, and hence the capacity of all the communication channels for this state. By traversing the finite set of reachable markings, we can compute the upper bound on the number of elements in each communication channel.

Code generation

The last step of the QSS algorithm is the generation of an efficient implementation of the schedule obtained in the previous phase. A direct translation of the schedule into C code would be possible, but the size of the code would be probably bigger. We will take advantage of the two facts. First, we know that possibly some code fragments will be replicated in different parts of the schedule; we can reduce the size of the generated code by generating a single copy of these code fragments. Second, we have established bounds for all communication channels inside the system. If the source and target of one channel have been merged in a sequential code, we can replace the communication channel by a circular buffer of known size, replacing read/write operations by read/write from the buffer. Moreover, if the buffer has size one, the communication channel can be replaced by a single variable, getting a huge speed-up in run-time and allowing compile-time optimizations of the code.

2.3.4 False path problem

During synthesis, data-dependent constructs are analyzed conservatively, in the sense that all possible outcomes of the constructed are considered feasible. However, there might be paths, called *false paths*, that are unfeasible at run-time, because the value of the variables prevents that path from being taken. In QSS, all these paths are being considered, something which lead to inefficiency, and in some extreme cases, to the impossibility to perform synthesis of systems due to huge number of false paths that have to be taken into account. Finding an automatic mechanism to prune these false paths automatically and statically is an open problem.

In section 4.3, we present our contribution towards the solution of this problem: an automatic technique for the static elimination of false paths based on abstract interpretation.

Chapter 3

Goals of the thesis

The goal of this thesis will be the research of efficient algorithms and data structures for the analysis of concurrent systems, based on the theory of abstract interpretation. The kind of analysis that will be explored are those with application to *verification*, *timing verification* or *synthesis* of concurrent systems. Also in the scope of this thesis, we will implement the proposed techniques into verification and synthesis tools and evaluate the efficiency and applicability of the proposed approaches.

The problem of data-flow analysis of concurrent systems has a very high computational complexity. Therefore, the main concerns in the research will be the *efficiency* of the proposed techniques and the *scalability* of the results, so that they can applied to analyze systems of a relevant size.

The following problems and research directions have already been identified:

Timing analysis of systems with symbolic delays

There are several techniques for computing conservative timing constraints for the correctness of concurrent systems. These techniques are based on analyzing the system with known constant delays.

Using abstract intepretation, it is possible to compute timing constraints for the system without having to specify delays, leaving *unknown* delays represented as *symbols*. This kind of constraints are very useful in the sense that they *characterize* the correctness of the system for any possible delay. For example, it is possible to see that the correctness of a circuit is independent of the delay of some of its components.

In this area, we plan to study algorithms for the timing analysis of concurrent systems with symbolic delays. The results of this analysis can be applied directly to the verification of timed asynchronous circuits. Part of this work has already been performed, and it is presented in Section 4.2.

Synthesis of embedded software

In the area of embedded system design, the synthesis of sequential implementations from concurrent implementations has a special relevance. Efficiency of the synthesized implementation is a major concern, so most of the work should be performed at compile-time. However, some problems, like the outcome of data-dependent choices, cannot be analyzed precisely at compile-time.

Abstract interpretation can solve this problem with approximate compile-time analysis of data-dependent choices. The results of this analysis can be used to predict the sizes of communicating channels, or detect paths of execution that are unfeasible due to the values of variables. Section 4.3 presents our contributions in that direction. There are additional benefits from applying abstract interpretation to the synthesis proces: the possibility of performing *performance estimation* and *verification* of the generated code.

Study of new abstractions of numeric values

In the literature of abstract interpretation, there are several abstractions that represent constraints on numeric variables. Each of them provide a unique trade-off between complexity and expressiveness of the supported constraints. However, there are some problems for which there is not an abstraction with a suitable trade-off.

For example, in timing analysis, it is often interesting to express that the delay of one execution path is smaller than the delay of another execution path, e.g. $d_1+d_3 \leq d_4+d_5+d_6$. These *path* constraints are linear inequalities where all coefficients are in $\{-1, 0, +1\}$. Clearly, these constraints can only be represented with convex polyhedra. However, the fact that all coefficients are $\{-1, 0, +1\}$ could be used to define a more efficient representation.

We will study the performance of state-of-the-art techniques and explore alternative representations of numeric constraints that provide better trade-offs than existing representations.

Efficient strategies for solving systems of equations

Abstract interpretation reduces the dynamic behavior of a system to a set of equations. Solving this system of equations is done iteratively until a fixpoint is reached. This computation usually has a very high complexity. Finding a way to speed up this computation can provide a huge impact in the overall efficiency of the analysis.

In this area, we plan to study different mechanisms to accelerate the process of finding a solution to the system of equations. Moreover, we intend to study techniques to perform analysis of subsets of the system of equations in order to get approximate results quickly without having to wait for convergence of the entire system of equations.

Decidibility and complexity aspects

Many interesting problems related to the static analysis of systems are undecidable, because they can be reduced to the halting problem of the Turing Machine. Abstract interpretation faces those problems by using approximation, i.e. for some problems, in addition to “yes” or “no”, the answer can be “I don’t know”.

In static analysis there are also many problems that are decidable through exact techniques. We plan to study the decidability and complexity of the problems solved using abstract interpretation techniques. For example, we are considering properties of a system that might be sufficient to ensure that exact analysis is decidible.

Compositional and hierarchical abstract interpretation

The results of static analysis techniques can provide very precise information about the state of a system. However, the complexity of such analysis makes it difficult to be applied to systems of a relevant size.

We will study strategies to divide the analysis of a complex system into several analysis of simpler subsystems and compose the results of the parts for the verification of the whole system.

Applicability of symbolic techniques

Decision Diagram techniques such as Binary Decision Diagrams have been applied successfully to many domains. The ability to manipulate large sets of states *symbolically* allows an efficient analysis of systems with very large state spaces, something which is very interesting for our domain of study. However, there is no symbolic representation which is fully adequate for abstract interpretation.

In our work in this area, we plan to study efficient abstractions of numeric values that can be represented using symbolic techniques, and symbolic versions of the abstract interpretation algorithm that rely on the previously defined abstractions.

Chapter 4

Contributions

4.1 Summary of the contributions

This chapter describes the original results that have been obtained in the area of verification and synthesis of concurrent systems. The main research results that are presented in this chapter are:

- An algorithm for the automatic discovery of *linear delay constraints* that guarantee the correct operation of a timed circuit.

  The relevance of this technique is based on the fact that the algorithm works with *symbolic* delays instead of requiring known constant delays as in previous works. Symbolic information can be used to implement the circuit more aggressively, reducing the delay of the circuit while ensuring that no error may occur. This algorithm has been implemented in a verification tool and it has been used to verify a benchmark of asynchronous circuits, including several patented circuits.

- A proposal of an automatic algorithm for the removal of false-paths in Quasi Static Scheduling.

  Quasi Static Scheduling requires exploring the possible execution paths that can be taken by a set of processes. Many of these paths are unfeasible because of the data values, e.g. loops may not iterate infinitely. All these paths have to be considered during the scheduling and increase the complexity of the scheduling procedure. Our proposal describes a pruning algorithm that analyses data values and remove the unfeasible (false) paths. This is the first fully automatic algorithm for the compile-time removal of false-paths.

The remaining of this chapter is organized in three blocks: the description of the timing verification algorithm; the description of the false-path elimination algorithm; and the list of the publications where these works have been presented.

4.2 Timing Verification of Concurrent Systems with Symbolic Delays

4.2.1 Introduction

The correctness of concurrent systems often depends on the temporal characteristics: response times, time-outs, computational delays, etc. Several formalisms have been proposed to model such systems, such as Timed Transition Systems [64], Timed Automata [5] and Hybrid Automata [4].

In these models, a system is specified as an automaton with annotated timing information. Given a property, verification usually gives an answer of this sort:

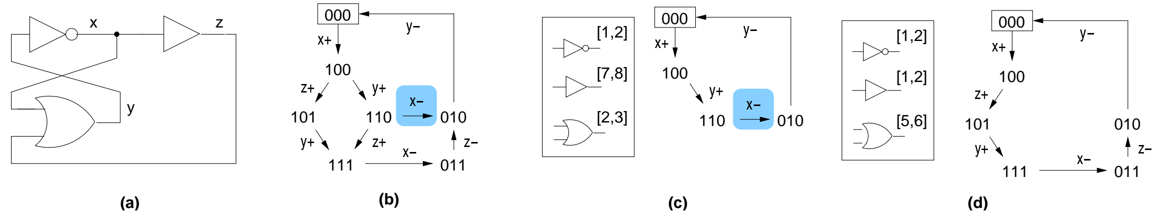


Figure 4.1: Motivating example: (a) an asynchronous circuit, (b) reachable states without considering delays, (c-d) reachable states with different sets of gate delays. The shaded area highlights the hazard of this circuit.

- *The system is correct* or
- *The trace σ leads to a failure*

where σ is a sequence of events annotated with time. However, the previous answer is only valid for the particular timing information provided for that instance of the system. Let us assume, for example, that this information is the set of gate delays of a circuit. The answer would only be valid for a particular technology, and could not be extrapolated to other technologies. Would it be possible to give a characterization of the circuit as a set of timing constraints that could guarantee the correctness of the circuit and that would be independent from the technology?. For example, the following answer would be much more meaningful: *The circuit is correct if*

$$\begin{array}{rcll} \delta(G_1) + \delta(G_2) & < & \delta(G_3) + \delta(G_4) + \delta(G_5) & \wedge \\ \delta(G_2) & < & 3\,\delta(G_6) + 2\,\delta(G_7) & \end{array}$$

where $\delta(G_i)$ denotes the delay of the gate G_i. The advantage of this type of answer is obvious. However, this requires an analysis with *symbolic* delays, that makes verification much more complex.

This section presents an algorithmic approach for the *automatic discovery of linear constraints in timed systems that guarantee their correctness*. One of the main motivations of this work is the characterization of the behavior of asynchronous controllers. The correctness of these circuits often depends on the actual delays of the gates. Under certain gate delays, the circuit may manifest hazardous behavior that can be propagated to some output signal and produce a failure. The purpose of the verification is to derive a set of linear constraints on the gate delays that guarantee a hazard-free behavior. Each constraint usually refers to a pair of structural paths in the circuit whose delays must be related by an inequality (e.g. delay(path_1) $<$ delay(path_2)).

The complexity of the problem restricts the size of the circuits that can be verified with this approach, since explicit representations of the states are required. So far, circuits with up to 15 symbols have been verified. This makes the approach specially suitable for the verification of small circuits whose behavior depends on the timing characteristics of the components, such as asynchronous controllers. Some examples of these controllers are the IPCMOS circuits from IBM [90], the self-resetting domino controllers used in the integer execution unit of the Pentium® 4 processor from Intel [65], or the GasP FIFO control circuits from Sun Microsystems [94].

4.2.2 A motivating example

Before describing the technique, we will discuss a motivating example from the domain of asynchronous circuit design. Asynchronous circuits do not use a global clock for synchronization, but additional design and synthesis effort is required in order to make sure that the implementation is free of *hazards* [78]. In a

discrete model of a circuit, a hazard can be conservatively modeled as a signal transition that *disables* a gate that was previously enabled, i.e. before a gate completes changing the output signal, input signals change to a state where the output signal no longer needs to be changed. In the analog world, a hazard can manifest a momentary pulse that can produce a circuit malfunction.

Figure 4.1(a) describes an asynchronous circuit with three gates. Figure 4.1(b) depicts the reachable states annotated with the signal values (x, y, z), starting from all signals at zero. In the untimed state space a hazard occurs in state 110, where the output of the buffer is zero, but it is changing to 1. If the inverter $(x-)$ completes faster than the buffer $(z+)$, then the buffer is disabled. However, this reachability graph does not take gate delays into account. The reachability of the hazard depends on the delays, being reachable for the delays in Figure 4.1(c) and unreachable for the delays in Figure 4.1(d).

The approach presented in this section aims at discovering the necessary restrictions that have to be satisfied by the delays to avoid all hazards, providing a more general result than those of non-symbolic techniques, that can only prove or disprove the existence of hazards for a set of known delays. An example of the constraints that can be computed automatically with this technique is:

$$D_{\text{buf}} < d_{\text{inv}} + d_{\text{or}}$$

where d and D represent the minimum and maximum delays of the gates. This restriction is sufficient to avoid the hazard in this circuit, so that *any* choice of delays that satisfies this property will not exhibit the hazard. The most interesting aspect of this characterization is that it is *technology independent.*

4.2.3 Timing Verification without Symbols

Several techniques for computing conservative timing constraints for the correct operation of asynchronous circuits are available in the literature. The main difference between this section and these approaches is that they are based on analyzing the circuit with known constant min-max delays in gates and wires [17, 26, 34, 71, 84]. The approach presented in this section can deal with *unknown* delays that are represented as *symbols.* Therefore, the analysis can be performed without making any assumption on the delay of the components of the circuit or the events of the environment.

There have been few contributions related to timing analysis with symbolic delays. In these contributions, Presburger arithmetics has been used for the analysis of timed systems with symbolic delays in several related problems. In [7], an application to the verification of timing diagrams with symbolic delays is presented. In [8], Presburger arithmetics is applied to the problem of *symbolic time separation of events*, which is vaguely related to timing verification. However, the efficiency of the technique presented is this section is superior to that of Presburger arithmetics regarding *both* the number of symbols that can be handled and the size of the systems that can be analyzed.

The kind of timing constraints that can be computed in other techniques also differs from our approach. The first class of constraints is *metric timing* constraints, i.e. constant min-max bounds for the components of a circuit. In [71], contraints are described as bounded delays called *delay paddings* that have to be introduced in the circuit to guarantee correctness. [26] computes delay paddings, plus the required delay bounds on input events. Another group of constraints is *relative timing* [17, 34, 70, 84], i.e. constraints that describe the relative order among concurrent events. Our approach can compute a wider class of constraints, *linear constraints.* Therefore, our analysis provides less conservative timing constraints, that can yield an increase in performance.

Our approach uses convex polyhedra as the abstraction to represent sets of timed states. In [50] convex polyhedra are used to analyze linear relations among variables, in the context of algorithms for the static analysis of programs. To preserve closedness in set operations, polyhedra can only represent approximations of the state space. For example, the union is not closed for convex polyhedra. As an overapproximation,

```
Algorithm AbstractInterpretation (G, A_In)
Input: A graph G = (N, E) with initial node In and initial constraints A_In.
Output: The abstraction Time for all nodes and edges.

  foreach node n ∈ N do Time(n) := ∅; endfor
  foreach edge e ∈ E do Time(e) := ∅; endfor
  Time(In) := A_In;
  changed := {N};
  do
    n := node in changed with lowest DFS number;
    changed := changed \ n;
    foreach edge n -e-> m ∈ E
      newTime := transfer(Time(n));
      if (newTime ⊆ Time(e)) continue;
      Time(e) := newTime;
      if (Time(e) ⊆ Time(m)) continue;
      if (e is a back edge)
        Time(m) := Time(m) ∇ (Time(e) ⋃ Time(m));
      else
        Time(m) := Time(m) ⋃ Time(e);
      changed := changed ⋃ {m};
  while (changed ≠ ∅);
```

Figure 4.2: Abstract interpretation algorithm

the convex hull is used instead. This strategy has also been used by other authors for the approximate verification of real-time systems [53], linear hybrid automata and synchronous programs with counters [63]. Linear hybrid automata and synchronous programs differ from timed transition systems in the condition required for an event to happen, i.e. there is no restriction on the time elapsed since an event becomes enabled for firing until it is finally fired, contrary to the lower and upper bound requirements defined in timed transition systems.

4.2.4 Timing analysis algorithm

Events of a TTS 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. Analysis of the values of event clocks can reveal whether an event can be fired or not in a given state.

This section presents an algorithm that computes a conservative upper approximation of the event clock values. Approximations will be propagated and combined using fixpoint techniques described in abstract interpretation.

Abstract interpretation for Timing Analysis

For the problem of timing analysis of a TTS, a configuration is a set of valid assignments of constant values to clocks and symbolic delays. We will abstract the set of valid assignments as a convex polyhedron that is an upper approximation of this set, i.e. all valid assignments are included in the polyhedron. The convex

polyhedron will describe the linear constraints that are satisfied among clock values and symbolic delays in all these valid assignments.

There will be two kinds of locations of interest of our timing analysis of TTS: states and transitions. We will note the abstraction in a given location x as $\mathsf{Time}(x)$, even though the abstraction has a different meaning for states than for transtions.

- In states, we are interested in the value of clocks when a state is reached, i.e. the *precondition* of the state.
- About transitions, we would like to compute the value of clocks after the transition happens, i.e. after firing an event. This can be considered as computing the *postcondition* of the transition.

In order to define the timing behavior of the system, we have to build a system of equations that defines how time elapses. When 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 in section 4.2.4) that advances the clock values while satisfying all upper and lower bounds. Using this function, the abstractions for states and transitions can be defined as the following system of equations:

- $\forall n \stackrel{e}{\rightarrow} m \in T : \mathsf{Time}(e) = transfer(\mathsf{Time}(n))$
- $\forall m \in T, n \stackrel{e}{\rightarrow} m \in T : \mathsf{Time}(m) = \bigcup \mathsf{Time}(e)$

Figure 4.2 describes an algorithm that computes a solution for this system of equations using a *increasing* fixpoint. Each location starts with an empty set of valid assignments to clocks and values, i.e. and empty abstraction. The algorithm applies the equations iteratively as long as the add new valid assignments. The solution is reached when there is a fixpoint, i.e. applying all equations another time does not yield any new states in any location of the system.

Termination, i.e. convergence of the system of equations, is guaranteed by using a widening operator ($A \nabla B$) for loops, as discussed in Section 2.1.3.

The clock transfer function

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, i.e. there is no restriction on their value. After firing an event, event clocks should be updated, because some time has elapsed since 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.
- Step should be long enough to ensure that the firing of e happens at least d_e time units after e was enabled. At the same time, it should be short enough to ensure that e is fired at most D_e time units after becoming enabled.

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

$P := P \wedge (step \geq 0)$;
$P := P \wedge (clock_e + step \geq d_e)$;
$P := P \wedge (clock_e + step \leq D_e)$;
`foreach` event $e' \neq e$: $e' \in \mathcal{E}(src)$
  $P := P \wedge (clock_{e'} + step \leq D_{e'})$;
`foreach` event $e' \neq e$: $e' \in \{\mathcal{E}(src) \cap \mathcal{E}(dst)\}$
  $P[clock_{e'} := clock_{e'} + step]$;
`foreach` event $e' \neq e$: $e' \in \mathcal{E}(dst) \wedge e' \notin \mathcal{E}(src)$
  $P[clock_{e'} := 0]$;
`foreach` event $e' \neq e$: $e' \in \mathcal{E}(src) \wedge e' \notin \mathcal{E}(dst)$
  $P[clock_{e'} :=?]$;
`if` $(e \in \mathcal{E}(dst))$ $P[clock_e := 0]$;
`else` $P[clock_e :=?]$;
$P[step :=?]$;
`return` P;

Figure 4.3: Clock transfer function

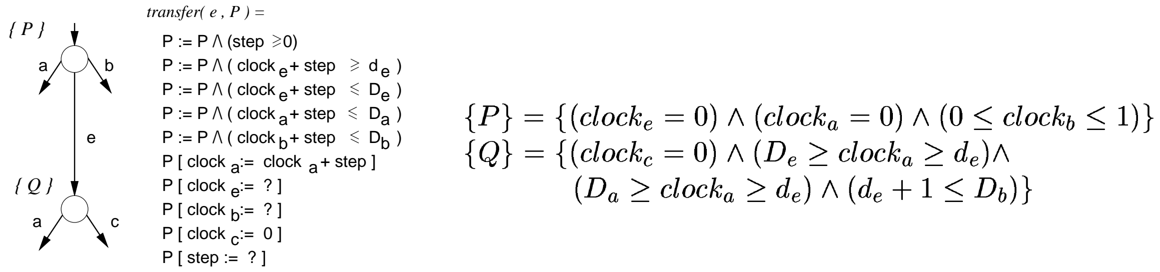


Figure 4.4: Example of the transfer function for an event e, with the postcondition Q obtained from a precondition P.

- Step should be short enough to ensure that any transition that is enabled before firing e is not forced to fire due to its upper bound constraint.

Once the clock step has been defined, the update in event clocks caused by the firing of an event e can be defined as:

- events that are disabled before and after firing e keep their clocks unchanged.
- events that are enabled before and after firing e have their event clocks increased by the clock step.
- events that become enabled by the firing of e have their clock set to 0.
- events that become disabled by the firing of e have their clock undefined.

Figure 4.3 describes the algorithm that computes the transfer function using convex polyhedra operators. Figure 4.4 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 $clock_a + step \leq D_a$, $clock_e + step \geq d_e$, $clock_a = 0$, $clock_e = 0$.

The clock transfer function described in this section can be easily modified to deal with symbolic timed automata instead of TTS. Checking location invariants and enabling conditions for transitions can be modeled as adding linear constraints to the polyhedron, and resetting clocks can be done with linear assignments, both of which are available operations on polyhedra. The transfer function for timed automata would be defined as (1) increase clocks by step, (2) check the source location invariant, (3) check the enabling condition of the transition, (4) reset clocks, (5) check the target location invariant and (6) undefine step.

4.2.5 Verification of safety properties using timing analysis

Timing analysis provides the required constraints for the reachability of the states and transitions of the TTS. However, we are looking for the complementary conditions, i.e. the conditions that render failures unreachable. Therefore, an algorithm is needed on top of timing analysis to extract selected constraints from those provided by abstract interpretation. This algorithm is presented in Figure 4.5.

The input is the specification of a TTS: a set of discrete variables, a transition relation, an initial state, and the delays of each event. Additionally, a predicate describing the failure states and an invariant of known delay constraints are also provided. The output is a set of sufficient constraints that ensure the absence of failures.

The first step is the calculation of the reachable state space using untimed depth-first reachability analysis. During this stage, failure edges and states will be identified. Also during this traversal, all back-edges of loops are identified and nodes are numbered in quasi-topological order; this order will be used to speed up convergence of the abstract interpretation analysis.

Timing analysis can then be performed on the TTS. The result of this step will be a polyhedron attached to each state and transition of the TTS, including the edges that lead to a failure. The polyhedron attached

```
Algorithm Verification (S, F, I)
Input: A specification of a TTS S, a predicate F describing failure states and transitions,
and a predicate I describing known restrictions on the symbolic delays.
Output: A set of constraints on the symbolic delays that is sufficient to avoid the failures
defined by F.

  G := ReachabilityAnalysis(S, F);
  constraints := I;
  do
     AbstractInterpretation(G, constraints);
     C := set of linear constraints required to reach a
        failure that are not implied by constraints;
     choose a linear constraint c from C;
     constraints := constraints ∧ ¬c;
  while (any failure is reachable ∧ constraints ≠ false);
  {constraints = false implies an unavoidable failure}
  return constraints;
```

Figure 4.5: Main algorithm for verification

to each of these edges describes constraints that are required to reach a failure. If any of these constraints is false, the failure will be unreachable. For example, if one polyhedron has the constraints,

$$(a \leq b + c) \wedge (e \geq f)$$

then, the constraint that makes sure that the failure is unreachable is the following disjunction

$$(a > b + c) \vee (e < f)$$

The algorithm proceeds by choosing one of these linear constraints at a time and adding it to the invariant. Currently, this choice is performed interactively, even though we have plans to automate this procedure. The verification continues until all failures have become unreachable or the invariant is *false*. A false invariant means “cannot find a constraint that makes the system correct”. It can happen if the system has an unavoidable failure or the algorithm cannot find sufficient constraints due to approximation. On the other hand, the algorithm might return the initial invariant, which means that no additional constraints are required for the correctness of the system.

Approximation in the analysis

The problem of computing the set of feasible clock and delay values is computationally expensive. This is the reason why we are using an approximate analysis technique, such as abstract interpretation, instead of trying to compute an exact solution. In the context of our problem, we calculate an *upper* approximation of the state space that guarantees no false positives in the verification of safety properties.

The source of approximation comes from the union of reconvergent paths. In case of acyclic reconvergence, the union is approximated by the convex hull. In case of cyclic reconvergence, the widening operator must also be used to guarantee the convergence of the algorithm. This technique is crucial when symbols are used to represent delays, as the number of iterations of a loop may depend on the actual delays of the components. Since the delays are symbolic, this number may be unknown.

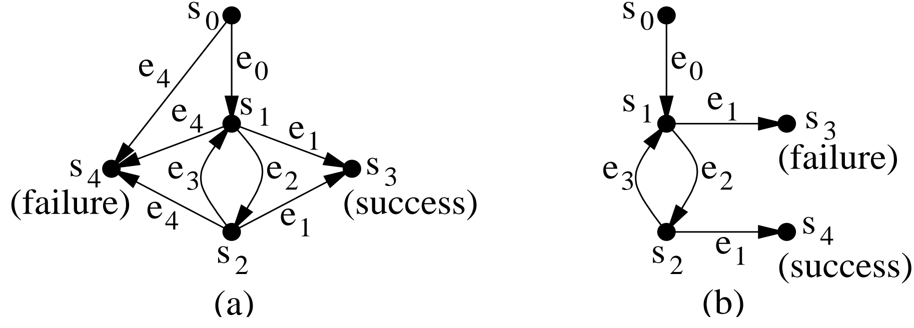


Figure 4.6: Cyclic behavior to illustrate the widening operator.

Figure 4.6 depicts two cyclic behaviors defined by the back edge $s_2 \xrightarrow{e_3} s_1$. Let as assume that each event e_i has a fixed delay $\delta(e_i)$. In Fig. 4.6(a), the correctness of the system is independent from the delays of e_2 and e_3. However, the absence of the widening operator would produce the following sequence of polyhedra in s_1:

iter.	Time(s_1)
0	$ck_{e_1} = ck_{e_2} = 0 \;\wedge\; ck_{e_4} = \delta(e_0) \leq \delta(e_4)$
1	$ck_{e_1} \leq \delta(e_2) + \delta(e_3) \leq \delta(e_1) \;\wedge\; ck_{e_2} = 0 \;\wedge$ $ck_{e_4} = ck_{e_1} + \delta(e_0) \;\wedge\; ck_{e_4} \leq \delta(e_4)$
$\ldots$	$\ldots$
i	$ck_{e_1} \leq i \cdot \delta(e_2) + i \cdot \delta(e_3) \leq \delta(e_1) \;\wedge\; ck_{e_2} = 0 \;\wedge$ $ck_{e_4} = ck_{e_1} + \delta(e_0) \;\wedge\; ck_{e_4} \leq \delta(e_4)$
$\ldots$	$\ldots$

where the predicate

$$ck_{e_1} \leq i \cdot \delta(e_2) + i \cdot \delta(e_3)$$

results from the convex union of the same predicate with equality instead of $\leq$, for all $0 \leq k \leq i$. With the widening operator applied after the first iteration, the polyhedron representing Time(s_1) would be reduced to

$$ck_{e_1} \leq \delta(e_1) \;\wedge\; ck_{e_2} = 0 \;\wedge\; ck_{e_4} = ck_{e_1} + \delta(e_0) \leq \delta(e_4)$$

This polyhedron would become invariant in the following iterations. After verification, the condition for absence of failure would be the following:

$$\delta(e_4) > \delta(e_0) + \delta(e_1)$$

Figure 4.6(b) depicts a situation of a *non-convex* condition for the avoidance of failures. It is easy to prove that the system is correct if the following predicate holds:

$$\exists i > 0 : i \cdot \delta(e_2) + (i-1) \cdot \delta(e_3) < \delta(e_1) < i \cdot \delta(e_2) + i \cdot \delta(e_3))$$

Unfortunately, the existential quantifier represents a disjunction that cannot be expressed as a convex polyhedron. In this case, the predicate for Time(s_1) would be:

$$ck_{e_1} \geq 0 \;\wedge\; ck_{e_1} \leq \delta(e_1) \;\wedge\; ck_{e_2} = 0$$

This abstraction does not show dependencies between symbolic delays. Therefore, the verification would not be able to provide any set of linear constraints to avoid the failure, even though there are values for delays that make the circuit correct.

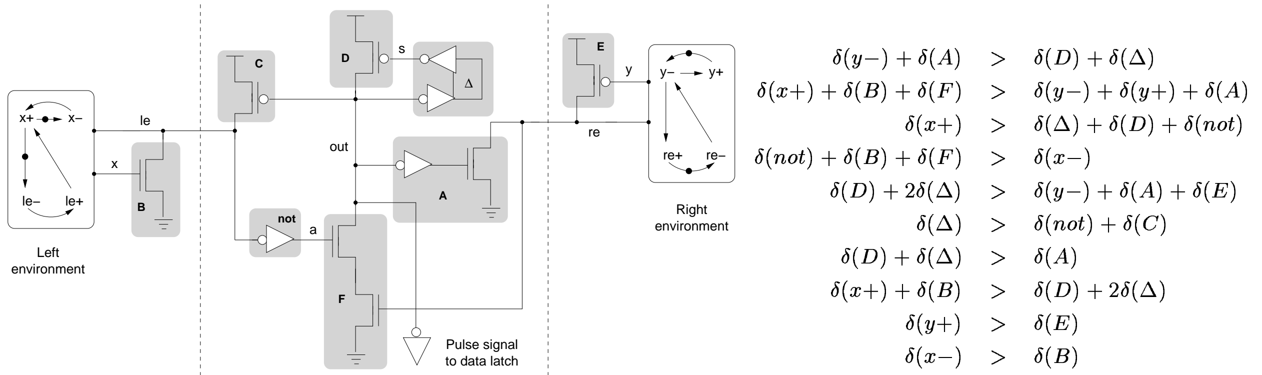


Figure 4.7: GasP FIFO controller. Each shaded area has been modeled with a different symbolic delay. On the right, the discovered timing constraints that are sufficient to guarantee the correct operation of the circuit.

4.2.6 Experimental results

In this section, we show some examples of asynchronous circuits that have been verified using the presented timing analysis technique.

GasP FIFO controller

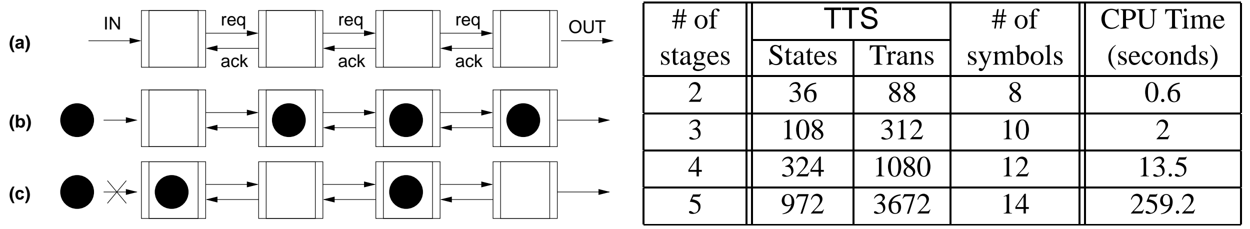
We have formally verified a GasP FIFO controller from Sun Microsystems [94]. This circuit handles the flow of data between stages of a pipeline: whenever the previous stage is FULL and the next stage is EMPTY, the control circuit (a) produces a pulse to the data latch in order to make it transparent, (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). Figure 4.7 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. The behavior of the environment is modeled with Signal Transition Graphs (STG) [30]. Environment events such as $x+$ or $y-$ describe the rising or falling of signals, and its delay models the time required to fire an event since it becomes enabled in the STG.

This asynchronous controller is designed to achieve a very high throughput, so it depends on timing constraints for its correct operation. In [70], this circuit is verified and sufficient relative timing constraints to ensure correctness are derived. However, it is hard to translate relative timing constraints into constraints on the delays of the components of the circuit.

The correctness of the circuit has been verified with respect to three criteria: *absence of short-circuits*; *absence of hazards*, i.e. once an event becomes enabled, it does not become disabled before being fired; and *conformance*, i.e. all output events produced by the circuit are expected by the environment. These criteria can be satisfied with the timing constraints that appear in Figure 4.7.

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"*. Figure 4.8 shows the pipeline, with an example of a correct and incorrect behavior.



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

Figure 4.8: (a) Asynchronous pipeline with N=4 stages, (b) correct behavior of the pipeline and (c) incorrect behavior. Dots represent data elements. On the right, the CPU times required to verify pipelines with different number of stages.

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. CPU time for the different lengths of pipeline can be found in Figure 4.8.

Other examples

We have also verified a set of asynchronous circuits available in the literature, defined as a network of simple gates plus a STG modeling the behavior of the environment. In these circuits, correctness has been defined as *absence of hazards* and *conformance* with the STG. Table 4.1 shows the size of the circuits, STGs and the computed TTSs, the number of symbolic delays, the number of constraints required for correctness, and the CPU time used for the verification.

Figure 4.9 shows an example of non-speed independent asynchronous circuit. Gate delays have been divided in the following categories: OR-gates ($[\delta, \Delta]$), 2-input AND gates $[\lambda, \Lambda]$), 2-input AND gates with an inverter ($[\omega, \Omega]$), 3-input AND gates with an inverter ($[\sigma, \Sigma]$) and environment events ($[\pi, \Pi]$). The restrictions on symbolic delays computed by our tool to ensure correct behavior are:

$$(\Sigma < \sigma + \pi + \delta) \wedge (\Lambda < \sigma + \pi + \delta)$$

4.2.7 Conclusions and future work

An algorithm for symbolic timing analysis of concurrent systems 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.

The technique is well suited for analyzing small-sized timed circuits such as asynchronous controllers. These circuits often operate at very high throughputs, and they heavily rely on stringent timing constraints to ensure a correct behavior. However, more complex circuits can also be verified if (a) they are analyzed at a higher level of abstraction or (b) some of the delays are defined as constant delays instead of symbols.

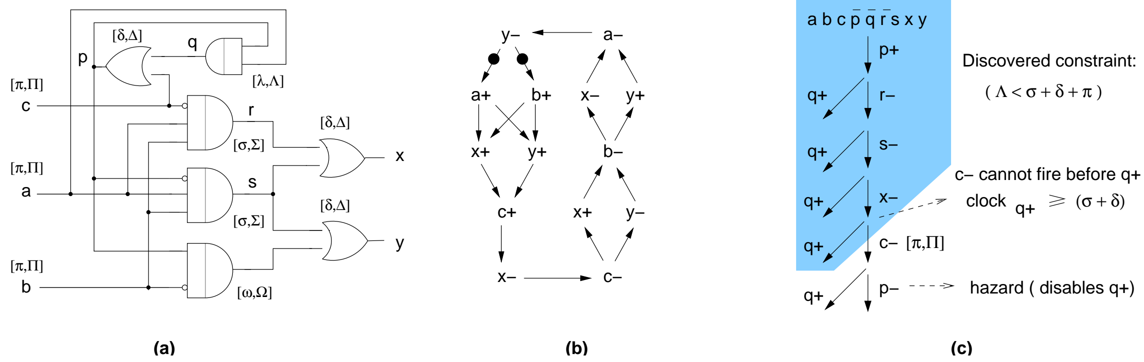


Figure 4.9: (a) Timed circuit for the `nowick` example, (b) STG modeling the behavior of the environment, and (c) example of a hazard trace avoided by the discovered constraints on symbolic delays.

Table 4.1: Experimental results

Example	Circuit		STG		TTS		# of	# of	CPU Time
	Signals	Gates	Places	Trans	States	Trans	symbols	constraints	(seconds)
nowick	10	7	19	14	60	119	10	2	0.5
gasp-fifo	9	7	10	8	66	209	12	10	8.1
sbuf-read-ctl	13	10	19	16	74	157	14	4	1.2
rcv-setup	9	6	14	15	72	187	12	8	2.1
alloc-outbound	15	11	21	22	82	161	19	3	1.3
ebergen	11	9	16	14	83	188	13	5	1.3
mp-forward-pkt	13	10	24	16	194	574	12	6	1.9
chu133	12	9	17	14	288	1082	7	3	1.3
converta	14	12	16	14	396	1341	14	13	20.4

Future work will try to broaden the area of application of this technique, in order to handle bigger circuits with more symbolic delays. We plan to use representations based on Binary Decision Diagrams to represent sets of states and timing constraints symbolically.

4.3 Scheduling of Concurrent Systems

4.3.1 Introduction

Embedded systems are driving research from the electronic community due to their widespread use. The design of embedded devices requires a multidisciplinar knowledge of software and hardware systems. It also requires a precise knowledge of how the system interacts with the environment.

The implementation of embedded systems is generally a software component running on top of a hardware architecture that might include several CPUs, DSPs, co-processors, and so on. The synthesis of this software-hardware mixed implementation defines three problems: (a) a *mapping* from elements of the specification to hardware and software components, (b) an *allocation* of software components to hardware computation units (e.g. CPUs) of the embedded system, and (c) a *scheduling* of the software components in each of the hardware computation units. Technology mapping and allocation are dependent on several factors, including cost and technology constraints. However, the remaining scheduling problem allows a formal approach as it is only dependent on the constraints dictated by the environment.

In the case of reactive systems, the scheduling problem is subject to timing constraints on the response to the inputs from the environment. Minimizing run-time of the input processing, if possible, and minimizing idle CPU time will the main concerns of the scheduling algorithm. The former is a matter of choosing the best algorithm and using an optimizing compiler. But the latter can be improved by reducing the CPU time spent by the operating system.

One possible scheduling approach could be *dynamic scheduling*: we generate one process for each functional process in our specification, letting the operating system decide which process should be executed at run-time. The problem with dynamic scheduling is that task context switching causes a run-time overhead, which can be unacceptable in some environments. Therefore, although dynamic scheduling is used widespread in interactive systems, it is not appropiate for reactive systems.

Static scheduling techniques try to make most decisions about execution order at compile-time, thus reducing the time consumed by the operating system during run-time. These algorithms require that some information about the rate of arrivals of inputs from the environment is known at compile-time. Even then, completely static schedules are only possible if we restrict ourselves to specifications without data-dependent choices. Real-time embedded systems, where a failure to satisfy a deadline can lead to a serious failure, can cope with this restrictions. But generic embedded systems would be very limited without data-dependent constructs. The problem of finding partial schedules in a concurrent system with data dependent constructs ensuring that there will not be buffer overflow is known as *Quasi Static Scheduling* (QSS).

However, the QSS algorithm does not analyze data values, and whenever a data-dependent construct is found, it is assumed *conservatively* that any branch of the choice could be taken. This decision forces the algorithm to explore all execution branches, even those *false paths* which are unreachable in practice. Considering false paths during the synthesis increases considerably the complexity of the QSS algorithm.

Several approaches have been presented in order to deal with this *false path problem.* In [11], a semi-automatic technique to deal with this problem is presented, which requires some interaction from the designer. In [93], an extension of the QSS algorithm that can deal with a more general class of specifications is presented. However, the synthesized code is multithreaded, so that part of the scheduling task is performed at run-time. However, both approaches fail to deal with the problem *both automatically and at compile-time*. This section presents briefly the QSS algorithm and our contribution, an automatic technique for the

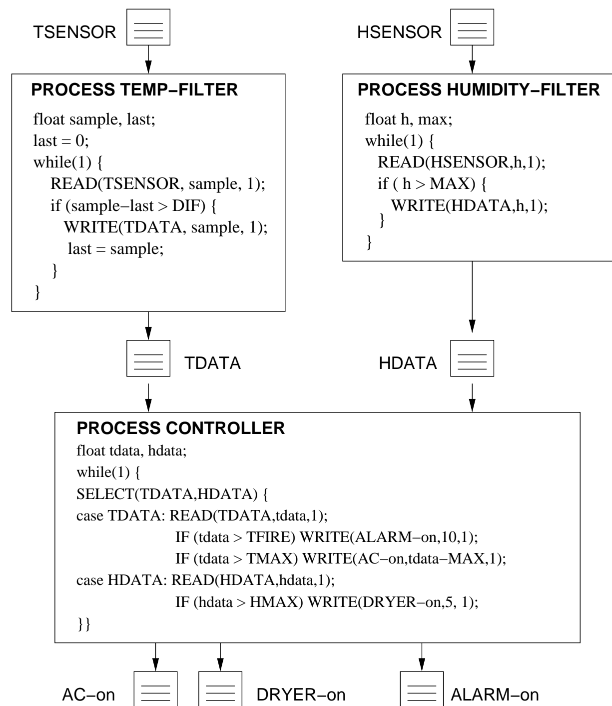


Figure 4.10: System specification example: environmental controller

compile-time elimination of false paths that is based on abstract interpretation.

4.3.2 Quasi-Static Scheduling

A motivating example

Figure 4.10 shows an example of embedded system that controls the environmental conditions in a room of a library. The books in the room can be damaged by a high humidity or a quick rise in temperature. Two sensors in the system monitor the temperature and humidity in the room; the system can react by activating air conditioning or a dryer. Finally, if the temperature becomes too high, the system assumes that there is a fire and raises an alarm. The system is divided in three processes. Two processes, namely the `TEMP-FILTER` and `HUMIDITY-FILTER`, filter the readings from the sensors that are not relevant to the system, such as low humidity or small change in temperature. A third process, the `CONTROLLER`, chooses the correct reaction to the current condition of the system. The communication among the three processes is performed using two internal ports, `TDATA` and `HDATA`.

The system is described using a language called *FlowC*. FlowC is basically a C language augmented with communication and synchronization constructs. The constructs related with communication specify the port where the information is read/written, the values read or written in these ports and the number of elements read or written: `READ(port, item, nitems)` and `WRITE(port, item, nitems)`. Communication has blocking semantics and it uses a FIFO policy. The third argument of `READ` and `WRITE` allows multi-rate communication among processes. Several `READ` (`WRITE`) constructs can use different values for `nitems` as long as the value is a constant known at compile-time. Synchronization is provided by means of a `SELECT` construct, that is used when the system is waiting for an input in a set of ports. The semantics is that when there is an input in a port or ports of the set, `SELECT` non deterministically chooses one of these available input ports.

A classical implementation would require three processes and an operating system that schedules their execution at run-time. We will analyze the results obtained from QSS and compare them with this implementation.

The QSS algorithm will generate a task for each input from the environment, i.e. there will be one task to process a temperature input and another one to process a humidity input. These tasks will be generated in three steps. First, a Petri Net model of the system will be built from the FlowC specification. Then, the Petri Net will be analyzed to find a set of run-time schedules that cover the run-time behavior of the system. Finally, we will generate code for these schedules.

Figure 4.11(a) shows the Petri Net generated by QSS from the environmental controller example. The code generated for the tasks in the system is displayed in 4.11(b). There is task for initialization, but it is negligible as it only has to be performed once at start-up. It is important to notice that, even though there are three processes in the original specification, there are only two tasks in the result of QSS, one for each input. The intuition for this is that each task performs the maximum amount of processing for the available input; although this sometimes implies a code replication among tasks, it reduces the amount of inter-task communication, because two tasks won't communicate unless two inputs are required to perform a computation.

The main advantage of a task-based implementation versus a process-based implementation is the reduction of the intervention of the operating system. A task-based implementation can use an interrupt based mechanism to wake up tasks when the system receives an input, which is much faster than the context switching required by a scheduling algorithm. The consequence is more time devoted to the execution of the system code and less time executing code from the operating system.

Figure 4.12 compares the process-based and task-based implementations. Task based implementation requires less interaction with the operating system, because of two reasons. The first one is that scheduling is not required. The second one is that processes require more internal communication to complete the task. Both factors justify our claims that a task-based implementation provides a much better performance than a

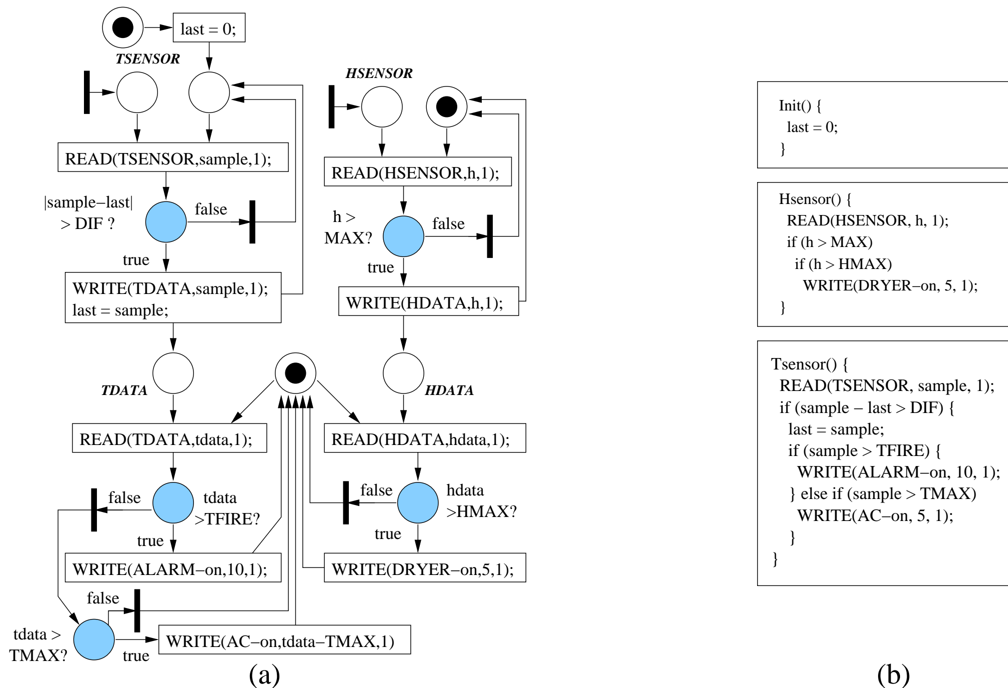


Figure 4.11: (a) Flow C specification, (b) Event-driven tasks after code generation

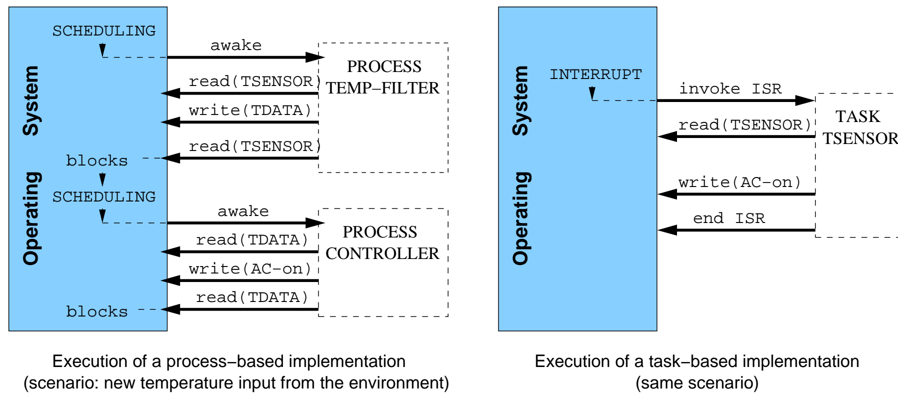


Figure 4.12: Comparison of task-based implementation versus process-based implementation.

process-based implementation.

Benefits and problems of the QSS approach

The QSS algorithm provides a mechanism to synthesize a task-based implementation from a concurrent specification. This implementation provides a better performance than a task-based implementation due to three factors:

- Reduction of the intervention of the operating system during run-time. In particular, the cost of run-time scheduling is reduced greatly as task context switching is not used if *all tasks use the same memory space* and *an interrupt-based mechanism is used to wake up the tasks.*

- Reduction of the memory requirements for our system. Communication buffers use up memory space, and they are required for inter-process communication. A task-based implementation implies less communication inside the system, so several of these buffers can be removed. Writing/reading from a buffer can be replaced by writing/reading from a scalar. For example, in Figure 4.11 channels `HDATA` and `TDATA` have been replaced by scalar variables.

- Improved chances of a good code optimization. As QSS replicates some code in several tasks, each task contains all the code related to the response to a particular input. Therefore, the compiler has new opportunities to perform code optimizations on the tasks, that were not available before due to the separation of processes. Moreover, replacing inter-process communication by scalar reads/writes makes the resulting code very adequate for data-flow analysis and optimization. Simple techniques like *copy propagation* and *dead-code elimination* [1] can noticeably improve the performance of the code of the tasks.

There are two main problems with the QSS approach.

- Replication of code among different tasks. This was considered a benefit, as it enabled further optimizations. However, it also has its drawback: the size of the code increases and therefore more memory is needed to store the code for the system. This problem is not so relevant, as time usually is much more critical than memory and reduction in memory usage by eliminating buffers makes up for this increase.

- Non-schedulability of some systems. The QSS problem has been proven to be undecidable. Our method copes with undecidability by accepting failures to generate schedules for some systems;

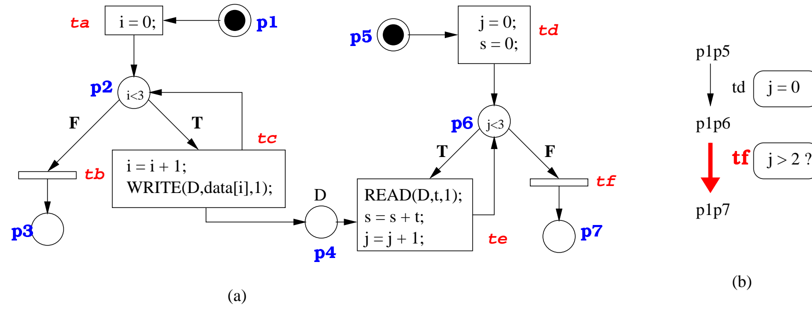


Figure 4.13: (a) Petri Net to be scheduled, (b) false path in the PN.

scheduling these systems might be possible by taking into account additional information that renders the problem undecidable. Finding a mechanism to schedule these systems is a very relevant issue.

4.3.3 False path detection

When the QSS algorithm is building the schedules, the data values are abstracted from the Petri Net, and data-dependent choices are abstracted as non deterministic choices. Whenever one of these data-dependent choices is reached, the QSS algorithm proceeds *safely*: it considers that *any* of the paths of the non-deterministic choice could be taken. However, some of these execution paths will not appear at run-time because of the data values of the variables. These execution paths that are considered by QSS (because data information is abstracted) but are unfeasible at run-time are called *false paths*. Figure 4.13 shows an example of false path in a PN schedule. The false path is a correct sequence of firings of enabled transitions, but transition `tf` will never be fired at run-time because the condition `j>2` will never be true at this point of the execution.

False paths can decrease the quality of the schedule generated for a PN or, even worse, can make the Petri Net non-schedulable. However, modeling data-dependent choices with the information of data values would render the problem undecidable, as it happens with Boolean Dataflow [24]. As an exact analysis is out of the question, the automation of this procedure can only be achieved through approximation. The proposed solution for automatic false path detection relies on the theory of abstract interpretation. Abstract interpretation offers a general framework for the static analysis of systems, that can be used to approximate the set run-time values of the variables of a program. Once the set of valid values of data is computed, this information can be applied to prune false paths each time that the synthesis algorithm reaches a data-dependent construct like `if` or `for`. From the theory of abstract interpretation, we know that the approximations will be a *safe overapproximation*. Therefore, even though a false path may not be discarded, a path will only be discarded if it is indeed a false path.

We have implemented an abstract interpreter that performs false path analysis on guarded transition systems that represent the reachable markings of the Petri Net. This abstract interpreter uses the convex polyhedra abstraction to compute the set of possible values of data. The values of data are used to prune false paths from the execution paths that can be taken, effectively simplifying the reachable markings for the QSS synthesis algorithm.

Figure 4.14 shows an example of a very simple system where false path elimination has been performed. The system is composed of two processes, where process 1 sends data to process 2 through a FIFO D, and process 2 simply accumulates the data recieved through the FIFO. The two processes can be translated

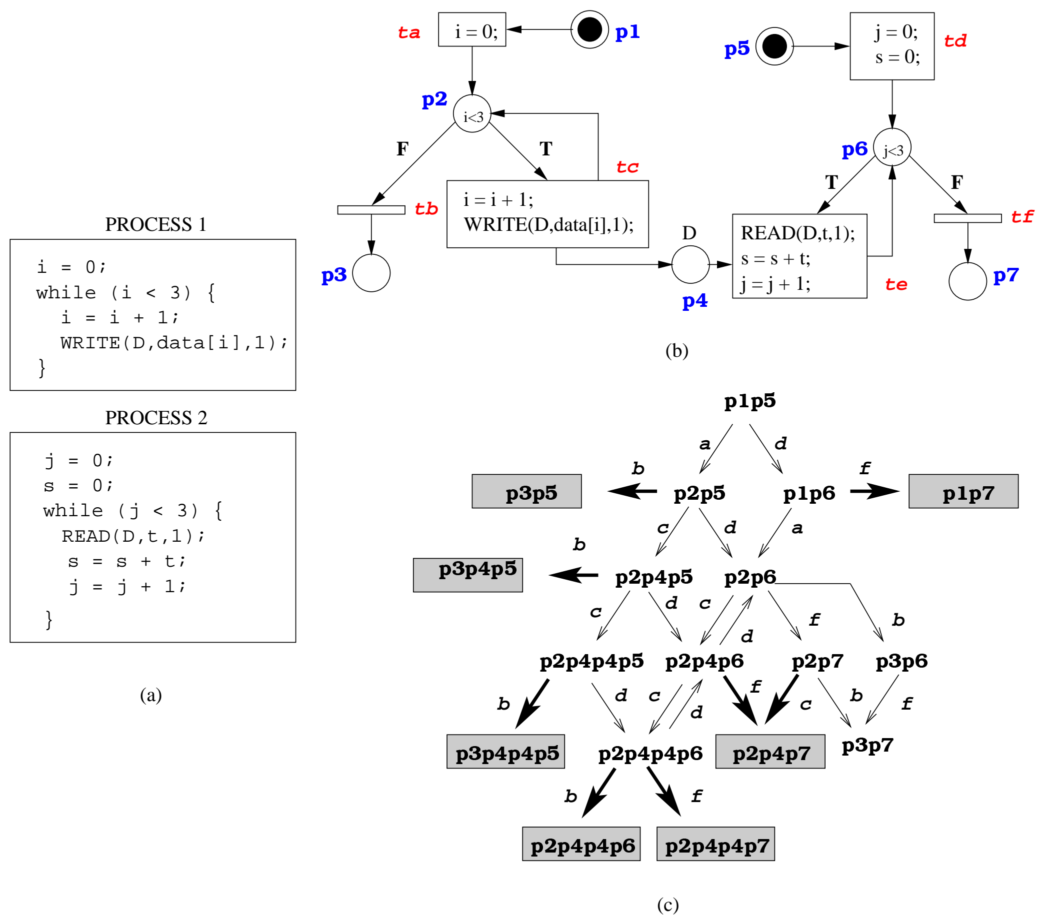


Figure 4.14: An example of automatic false path elimination: (a) Original specification of the system, (b) Petri-Net modeling the specification, (c) Reachable markings of the Petri-Net, where the grayed markings are false paths found automatically by the abstract interpreter.

into the Petri Net (b). At this point, reachability analysis combined with abstract interpretation yields the reachability graph in (c). Several markings have been detected as non-reachable without any kind of user interaction. Notice that the abstract interpreter has discovered several non-trivial properties that did not appear in the original program, e.g. the process 2 cannot terminate until the FIFO is empty.

4.3.4 Conclusions and future work

A technique for the elimination of false paths has been presented. Remarkably, the approach can be fully automated and performs the necessary analysis at compile-time. This technique can be used to improve the applicability of QSS to larger examples with data-flow dependent constructs. Experiments with the prototype implementation demonstrate the suitability of abstract interpretation for this kind of problem. Further experiments with real-world examples are required in order to prove the efficiency of this technique in larger systems.

Further research in this area will target other abstract interpretation analysis that can be very useful for the synthesized code, such as verification of the synthesized code or performance estimation.

4.4 Publications

The work discussed in this document has been presented in the following scientific publications:

- Robert Clarisó, Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Claudio Passerone and Yosinori Watanabe. Synthesis of Embedded Software for Reactive Systems. In *2nd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT'2002, Satellite Event of ETAPS 2002)*, pages 2–20, April 2002.

- Robert Clarisó and Jordi Cortadella. Symbolic timing analysis for the verification of asynchronous circuits. In *3rd Workshop of the Working Group on Asynchronous Circuit Design (ACID-WG'2003)*, January 2003.

- Robert Clarisó and Jordi Cortadella. Verification of Timed Circuits with Symbolic Delays. In *12th International Workshop on Logic and Synthesis (IWLS'2003)*, May 2003.

- Robert Clarisó and Jordi Cortadella. Verification of Timed Circuits with Symbolic Delays. Submitted to *IEEE/ACM International Conference on Computer Aided Design (ICCAD'2003)*, November 2003.

Chapter 5

Future work of the thesis

5.1 Remaining tasks

Chapter 3 presented the goals of the thesis and Chapter 4 has presented the results that have already been obtained. The remaining work consists basically on finding theory, efficient techniques and or data structures that improves the efficiency and scalability of abstract interpretation analysis.

Although the exact solutions to be proposed are still being researched, the following is a list of topics are going to be explored. This list should not be considered as the precise list of contents of the thesis. The research of the thesis will try to cover a significant part of these open problems.

1. Timing analysis of systems with symbolic delays
   (a) Study the extensibility of the proposed results to formalisms like Timed Automata or Hybrid Systems.
2. Synthesis of embedded software
   (a) Implement the proposed analysis algorithm into the Quasi-Static Scheduling framework and evaluate its efficiency with real-world examples like an MPEG-2 decoder [11].
   (b) Study the extensibility of the proposed analysis to provide performance estimations of the generated code.
3. Study of new abstractions of numeric values
   (a) Study a data structure for the representation of linear constraints with coefficients $\{-1, 0, +1\}$ to be used in timing analysis. An initial proposal of such a representation has already been made.
   (b) Implementation the previous data structure using Binary Decision Diagrams.
   (c) Evaluate the efficiency of this abstraction compared to less expressive representations, such as Difference Bound Matrices, and more expressive representations, such as convex polyhedra and Presburger arithmetics.
   (d) Study other implementations based on decision diagram techniques. In particular, study numerical representations such as NDDs or DDDs and their suitability to represent numeric constraints.
4. Efficient strategies for solving systems of equations

(a) Characterize properties of systems that permit more efficient solving strategies. For example, acyclic systems can be analyzed much more efficiently than other systems. The goal would be finding other non-trivial properties.

(b) Study how to divide the solution of the system of equations into smaller subproblems whose solutions can be combined. As an example, study how the results of the analysis of a system are related to those that can be obtained by analyzing the underlying spanning trees or directed acyclic graphs.

(c) Analyze the paper of the widening operator in abstract interpretation analysis, and the possibility of alternative implementations or

5. Decidibility and complexity aspects

   (a) Study the decidibility and complexity of the problem "check a LTL formula in a restricted class of TTS with symbolic delays". This problem requires exact timing analysis of TTS without any kind of approximation, i.e. without using the widening operator or the convex hull".

6. Compositional and hierarchical abstract interpretation

   (a) Study the hierarchical analysis of complex systems that are divided in modules whose communication is delay-insensitive. Reachability analysis of the whole system could be performed using symbolic techniques, while local analysis of each of the components would be performed by the classic abstract interpretation algorithm.

   (b) Study the composition of the results of abstract interpretation. Given a set of components, which have already been analyzed, study how the analysis results of one component can be applied to the whole system.

7. Applicability of symbolic techniques

   (a) Study how to use symbolic techniques in the reachability analysis performed before applying abstract interpretation.

   (b) Study how to implement the abstract interpretation analysis symbolically using, for example, the abstraction proposed in 2(b). This technique would combine with the symbolic reachability analysis in 5(a) to allow a fully symbolic abstract interpretation analysis.

   (c) Evaluate the performance of symbolic techniques applied to abstract interpretation.

5.2 Working plan

It is expected that the tasks related to the thesis work will be completed in two years. However, this is just a broad estimation. The duration of the thesis depends on the results that will be obtained during the evaluation of the proposed techniques. Most approaches in the area of verification have very bad worst case complexity, but then are widely used due to very good average case complexity. Some of the techniques that are going to be studied fall into this category, so its relevance depends on the efficiency that they exhibit in practical examples.

The planned schedule for the thesis is as follows. Each of the areas presented in the previous sections will be studied for a period of three months. In those areas, a set of relevant problems has already been identified. Depending on the results obtained in these problems, or new open problems that might arise, the period of time for each of these areas will be modified. The last six months will be dedicated to writing the thesis.

In this working plan, we have considered the work already done. During these initial years, the theory and applications of abstract interpretation have been studied in depth. Several examples that can be used as benchmarks for new analysis have also been identified. Finally, a generic library for abstract interpretation analysis has been developed. Any future implementation of new analysis techniques will be accelerated by the use of this library.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques and Tools*. aw, 1986.

[2] R. Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford University, Aug. 1991.

[3] R. Alur. Timed automata. In *Proc. International Conference on Computer Aided Verification*, pages 8–22, 1999.

[4] 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.

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] Y. Ameur, P. Cros, J.-J. Falcon, and A. Gomez. An application of abstract interpretation to floating-point arithmetic. In *Proceedings of the Workshop on Static Analysis*, 1992.

[7] T. Amon, G. Borriello, T. Hu, and J. Liu. Symbolic timing verification of timing diagrams using presburger formulas. In *dac*, pages 226–231, jun 1997.

[8] 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.

[9] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Computer Aided Verification*, pages 419–434, 2000.

[10] A. Arnold. *Finite Transition Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[11] G. Arrigoni, L. Duchini, L. Lavagno, C. Passerone, , and Y. Watanabe. False path elimination in quasi-static scheduling. In *Proc. of Design, Automation and Test in Europe*, pages 964–971, Mar. 2002.

[12] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. Quaderno 312, Dipartimento di Matematica, Università di Parma, Italy, 2003.

[13] 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/.

[14] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. *Hardware-software codesign of embedded systems: the Polis approach*. Kluwer Academic Publishers, 1997.

[15] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Y. Watanabe, and G. Yang. Concurrent execution semantics and sequential simulation algorithms for the Metropolis Meta-Model. In *International Symposium on Hardware/Software Codesign*, pages 211–216, 2002.

[16] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification*, pages 341–353, 1999.

[17] W. J. Belluomini and C. J. Myers. Timed circuit verification using TEL structures. *IEEE Transactions on Computers*, 20(1):129–146, 2001.

[18] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[19] 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. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, Oct. 2002.

[20] 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.

[21] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735:128–141, 1993.

[22] O. Bournez and O. Maler. On the representation of timed polyhedra. In *Automata, Languages and Programming*, pages 793–807, 2000.

[23] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1998.

[24] J. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model.* PhD thesis, U.C. Berkeley, 1993.

[25] J. R. Burch. *Trace algebra for automatic verification of real-time systems.* PhD thesis, Stanford University, Aug. 1992.

[26] 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.

[27] 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.

[28] 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.

[29] M. Chiodo, P. Guisto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. L. Sangiovanni-Vincentelli, and E. Sentovich. Synthesis of software programs for embedded control applications. In *Design Automation Conference*, pages 587–592, 1995.

[30] T.-A. Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications.* PhD thesis, MIT, June 1987.

[31] 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.

[32] T. Coe, T. Mathisen, C. Moler, and V. Pratt. Computational aspects of the Pentium affair. *IEEE Computational Science & Engineering*, 2(1):18–31, 1995.

[33] C. Colby. Determining storage properties of sequential and concurrent programs with assignment and structured data. In *Static Analysis Symposium*, pages 64–81, 1995.

[34] 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 of Integrated Circuits and Systems*, 21(2):109–130, 2002.

[35] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. L. Sangiovanni-Vincentelli. Task generation and compile-time scheduling for mixed data-control embedded software. In *Design Automation Conference*, pages 489–494, 2000.

[36] L. A. Cortés, P. Eles, and Z. Peng. A Petri net based model for heterogeneous embedded systems. In *Proc. of NORCHIP Conference*, pages 248–255, 1999.

[37] 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.

[38] P. Cousot. Abstract interpretation: Achievements and perspectives. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*. Scuola Superiore G. Reiss Romoli, July 2000.

[39] 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.

[40] 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.

[41] 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 SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, 1977.

[42] 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.

[43] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[44] P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, 1984.

[45] P. Cousot and R. Cousot. Relational abstract interpretation of higher-order functional programs. JTASPEFL '91. *BIGRE*, 74:33–36, Oct. 1991.

[46] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[47] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[48] 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.

[49] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the International Conference on Computer Languages*, pages 95–112. IEEE Computer Society Press, May 1994.

[50] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, New York, 1978.

[51] G. Dantzig and B. Eaves. Fourier-motzkin elimination and its dual. *Journal of combinatorial theory*, 14:288–297, 1973.

[52] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[53] 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.

[54] 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.

[55] 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.

[56] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validations and synthesis. *Proceedings of the IEEE*, 85(3), 1997.

[57] S. A. Edwards. Compiling concurrent languages for sequential processors. Technical Report CUCS-013-01, Columbia University, 2001.

[58] K. Fukuda. Frequently asked questions in polyhedral computation. http://www.ifor.math.ethz.ch/ fukuda/polyfaq/polyfaq.html.

[59] E. Gawrilow and M. Joswig. polymake: an approach to modular software design in computational geometry. In *Proc. of the 17th Annual Symposium on Computational Geometry*, pages 222–231. ACM, 2001. Library available at http://www.math.tu-berlin.de/polymake/.

[60] P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[61] 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.

[62] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[63] 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.

[64] 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.

[65] G. Hinton, M. Upton, D. Sager, D. Boggs, D. Carmean, P. Roussel, T. Chappell, T. Fletcher, M. Milshtein, M. Sprague, S. Samaan, and R. Murray. A 0.18-μm CMOS IA-32 processor with a 4-GHz integer execution unit. *IEEE Journal of Solid-State Circuits*, 36(11):1617–1627, Nov. 2001.

[66] C. A. Hoare. Communicating sequential processes. *International Series in Computer Science*, 1985.

[67] 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.

[68] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[69] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, Aug. 1974.

[70] H. Kim, P. Beerel, and K. Stevens. Relative timing based verification of timed circuits and systems. In *Proc. 8th Int. Symp. on Asynchronous Circuits and Systems*, 2002.

[71] 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.

[72] E. A. Lee and D. G. Messerchmitt. Static scheduling of synchronous data flow graphs for digital signal processing. *IEEE Transactions on Computers*, Jan. 1987.

[73] B. Lin. Software synthesis of process-based concurrent programs. In *Design Automation Conference*, pages 502–505, June 1999.

[74] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *Proc. ACM Int. Conf. on Supercomuting*, pages 226–235, 1992.

[75] A. Miné. oct: The Octagon Abstract Domain Library. http://www.di.ens.fr/ mine/oct/.

[76] 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.

[77] 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.

[78] D. Muller and W. Bartky. A theory of asynchronous circuits. In *Proc. International Symposium on the Theory of Switching*, pages 204–243, Cambridge, 1959. Hardware University Press.

[79] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989.

[80] New Polka: Convex Polyhedra Library. http://www.irisa.fr/prive/bjeannet/newpolka.html.

[81] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[82] B. Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, 1997.

[83] D. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, July 1978.

[84] 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, 2000.

[85] Z. Peng. *A formal methodology for automated synthesis of VLSI systems.* PhD thesis, Linköping University, 1987.

[86] C. Piguet et al. Memory element of the Master-Slave latch type, constructed by CMOS technology. US Patent 5,748,522, 1998.

[87] Polylib: A library of polyhedral functions. http://www.irisa.fr/polylib/.

[88] PolySpace Technologies. http://www.polyspace.com.

[89] Qhull: a library for computing convex hulls. http://www.geom.uiuc.edu/software/qhull/.

[90] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous Interlocked Pipelined CMOS Circuits Operating at $3.3 - 4.5$GHz. In *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, pages 292–293, Feb. 2000.

[91] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal models for embedded system design. *IEEE Design&Test of Computers*, 17(2), 2000.

[92] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, Aug. 2001.

[93] F. S. Su and P. A. Hsiung. Extended quasi-static scheduling for formal synthesis and code generation of embedded software. In *International Symposium on Hardware/Software Codesign*, pages 211–216, 2002.

[94] 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.

[95] US General Accounting Office report (GAO/IMTEC-92-26). Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia, 1992.

[96] M. Varea and B. Al-Hashimi. Dual transitions petri net based modelling technique for embedded systems specification. In *Proc. of Design, Automation and Test in Europe*, Mar. 2001.

[97] 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.