MASTER IN COMPUTER SCIENCE

Automata oriented program verification

Grégoire, Bertrand

Award date:
2002

FUNDP
Institut d'Informatique

Rue Grandgagnage, 21
B - 5000 NAMUR (Belgique)

Automata Oriented
Program Verification

Bertrand GRÉGOIRE

Under the advisory of Prof. Pierre-Yves Schobbens

Institut d'Informatique
Facultés Universitaires Notre-Dame de la Paix
Namur

Septembre 2002

Abstract

The formal verification process of a design with respect to temporal specifications is essential while designing (reactive) systems.

The process we present here relies on timed Live Sequence Charts (LSCs) to describe properties of the system under development. The LSCs are unwound into timed automata and automatically verified with a fair CTL model checker. We describe all the transitions that take part in this completely automatic process.

To obtain a more efficient verification we introduce a new class of automata, which is proven to be an interesting subset of the timed Büchi automata, and show, theoretically and algorithmically, how the efficient verification of these properties can be conducted using the VIS model checker.

Keywords formal verification, model checking, reactive system, temporal requirement, VIS, invariant check, timed automaton, non-failure acceptance.

Abstract

Il est essentiel, lors de la mise en place de systèmes réactifs, de pouvoir vérifier des propriétés temporelles dès les premières phases du développement.

Le processus de vérification automatisé que nous décrivons ici se base sur des Live Sequence Charts (LSCs) pour décrire les propriétes attendues. Ces LSCs sont applatis en automates temporisés, et la propriéte est vérifiée automatiquement par un vérificateur de modèles CTL équitable. Nous décrivons les traductions intermédiaires qui permettent cette vérification automatique.

Pour obtenir un processus plus efficace nous présentons une nouvelle classe d'automates, définie comme un sous-ensemble intéressant des automates temporisés de Büchi. Nous montrons comment une vérification efficace peut être conduite sur ces propriétés, en théorie et par une procédure concrète.

Mots-clés vérification formelle, vérification de modèles, système reactif, comportement temporel, VIS, vérification d'invariant, automate temporisé, acceptation sans-échec.

Acknowledgements

I would like to thank Prof. Pierre-Yves Schobbens, Mr. Patrick Heymans and Mr. Yves Bontemps for giving me the opportunity to discover the world of research, guided with their great insight. They were available to answer my questions and their (many) remarks greatly helped me in my work.

Prof. Bernhard Josko welcomed me in his Embedded Systems departement, at Oldenburg. There, I collaborated with Hartmut Wittke, Jocken Klose and Tom Bienmüller on the subject presented here. I would like to especially thank them all for their help, in Oldenburg or during the writing of this thesis. They found place for me in their overfull agendas, and made a place for me at their table. Thanks to Ingo, Alexander, Thomas and Ulf for having made my stay and work easier.

My friends have made my time in Oldenburg fun, thanks Christian, Céline, Nadine, Mad, Victor, Marina, Tomas, Karer, Celia, Rebecca, Shantala, Udo, Ditza, Frank, Fathi and the many others. Annika gave me her Germany, Louis his energy and Imke her thousands different smiles.

I owe much of who I am to my parents. They have started me on this path, and have always been with me along the way. For all the support they gave me, I would like to thank wy whole family: Valérie, Alex, maman and papa. Thanks also to Opa for the funny time we spent together in Germany, and to nonkel Luc, for his valuable pieces of advice about my bad english.

Finally, I would like to thank my girlfriend, Barbara. She forgave me the time we couldn't spend together and encouraged me every time I needed it, even without saying a word.

Contents

List of Figures

Introduction

The news group comp.risk is full of funny stories, such as the one reported in [Mur90], a British news paper. A runaway train went down the London's Tube track, leaving its driver standing behind, on the platform. The man actually left the cab of his fully-automated train to check a door which had failed to close properly. When the door did shut an electrical circuit was completed and the train, with 20 passengers on board, moved off before the driver had time to rush back to the controls. None where killed nor injured, but the driver has been sacked.

According to a U.S. Army report, a software problem contributed to digging holes at Fort Drum, in June 2002. Two soldiers were firing artillery shells, relying on the output of the Advanced Field Artillery Tactical Data System. But if one forget to enter the target's altitude, the system assumes a default of 0, when (part of) Fort Drum is at 679 feet above sea level. The report goes on to warn that soldiers should not depend exclusively on this one system, and should use other computers or manual calculations.

Software failures, and so are these unexpected behaviors, are a nightmare of many major firms. Let us just remember the paranoia we faced with the so called "Year 2000-Related" computer failures.

Unfortunately many human lives rely on software or hardware systems, which control airplanes, automobiles, nuclear power plants and medical laboratories, among others. These systems are called *safety-critical systems.* This designation regroups computer, electronic or electromechanical systems whose failure may cause injury or death to human beings.

Transformational and reactive systems

Most safety-critical systems are highly *reactive*, meaning they interact with their environment. The systems which are not reactive are transformational, we illustrate both systems behaviors in figure 1.

Transformational systems are those which have all inputs ready when invoked and the outputs are produced after a certain computation period. Most industrial processes are transformational systems, but a simple procedure that computes the square root of a number is a transformational process as well.

The reason a *reactive* system exists is typically to collaborate or interact with some entities in its environment. Sending, receiving, recognizing and subjecting sequences of symbols are parts of a reactive behavior. A well understood reactive system is a traffic-light controller. It is virtually impossible to write a transformational program that implements such a controller, since the inputs occur when the system is already running. In fact, most controllers are by definition reactive, with application domains ranging from process control, military, aerospace, and automotive applications to medical electronics, and similar embedded systems.

Amir Pnueli [Pnu77] calls "reactive systems" any nonterminating or continuously operating concurrent programs, such as operating systems or network protocols. The use of such systems is growing year after year. Graphical user interface based software (GUI) and embedded systems are typically reactive. The latter is often implemented as hardware.

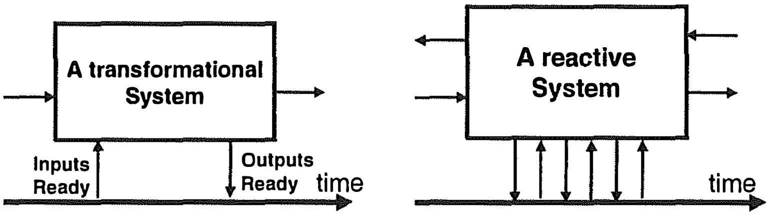


Figure 1: Transformational and reactive systems

Formal methods

The design error problem of life-critical (reactive) systems is a great threat to the human being. There are 3 basic strategies [Hol97] for dealing with design errors:

1. *Testing* (lots of!).

   The problem with life testing is that in order to measure ultra-reliability one must test for exorbitant amounts of time.

2. *Design diversity* (fault-tolerant software).

   The basic idea is to use separate design/implementation teams to produce multiple versions from the same specification. Then, non-exact threshold voters are used to mask the effect of a design error in one of the versions. The underlying hope is that the design flaws will manifest errors independently, or nearly so. In fact, design diversity can create an "illusion" of ultra-reliability, without actually providing it.

3. *Fault avoidance* (formal specification and verification, reusable modules)

   Formal methods may be used to *specify* and *model* the behavior of a system and to mathematically *verify* that the system design and implementation satisfy system functional and safety properties.

The often targeted $1-10^{-9}$ reliability is beyond the range of quantification (for both testing and design diversity) as stated in [Hol97]. We have hence no other choice than to develop safety-critical systems in the most rigorous manner available to us, which is the use of formal methods. We use the term *formal methods* to refer to the variety of mathematical modeling techniques that are applicable to computer system (software and hardware) design.

The formal specification challenge

Designers of today's reactive systems face design challenges of enormous complexity due to the increase of design content, explosion of features, ambiguous design parameters and evolving customer requirements.

A high-level (formal) specification written in a language that has a well defined semantics is mandatory for such designs. Nevertheless many companies still use hand-written (informal) specifications. Benefits of a formal specification compared with an informal one are very similar to the benefits of a true, working, program compared with a document describing what the program should do. The first is a usable object, while the later is nothing but a nice piece of paper that needs to be used by a human. Non formal methods of communication often leads to costly design and debug iterations [BF93]. Numerous studies have shown that correcting an error during integration costs over 10 to 1000 times more than correcting it at specification time.

The challenge of generating complete and unambiguous specifications is, of course, only half the battle. The other big unknown is whether the specifications accurately reflect user requirements [Gil97].

The formulation of requirements into a formal specification is called *synthesis.* Traditionally, methodologies were used to produce the model, based on experience in industry [BP94]. More recently, development environments provide automatic specifications synthesis, based on scenarios [I-L00b].

The formal verification challenge

Once we have got a formal representation of both the design under development and the requirements (specifications) we are excepting from it, it is time to check whether the first fulfills the later. This problem is called the *verification*, or *validation* process.

Many techniques have been studied for the past twenty years, developing a wide range of approaches and subsequent tools. Notations and languages that were previously designed by and for mathematicians are now adapted to the engineer's needs. The user interfaces are a major concern of most of the publicly available tools, especially the commercial ones. One step after the other formal verification methods have been adopted within the hardware community. Increasing cooperation between industry and formal methods researchers give rise to practical and even more efficient formal verification approaches.

Recent discoveries allow us to verify always larger designs with always more detailed properties. Complete formal verification of large complex systems becomes practical at this time, and a great increase in confidence in the system can be obtained by the use of formal methods at key locations in the system.

We do not assert formal methods are the silver bullet[1] that can magically lay all our problems to rest, citing Frederick P. Brooks, Jr. [Bro86]. Formal methods are rather a complement to good design methodology and testing.

The techniques of *automated* verification developed widely for the past two decades, and we try make a contribution. Our work investigates how formal proofs can be done on designs under development, and we try to make a contribution by improving some steps of this complex process, focusing on the translation of the specification set to allow more efficient verification methods to be used.

Master thesis structure

In chapter 1 we present the formalism we use to describe the properties we want to verify, namely *Live Sequence Charts* (LSC) [DH98]. LSCs are used to describe the interactions between many components of a reactive system within one scenario. They provide a means to distinguish between mandatory and possible behaviors of the components. We translate these charts into timed automata, to enable the automatic verification of the specification. Therefore we introduce a timed automata formalism and describe the translation of LSCs into such automata.

Chapter 2 describes how the problem of formal verification can be reduced to the problem *model checking* [CGP99]. We therefore describe the most widely used logics and techniques to automatically verify properties. Other approaches than model checking are evoked, as the automata-theoretic approach, for instance. An overview of more recent techniques, which are currently used in the model-checking field, is given as well.

Chapter 3 can be seen as the application of the previous chapter. After a quick survey of some well-known verification tools we apply the previously cited formal verification techniques into a real environment. The VIS model-checker, developed at Berkeley and Boulder, is investigated into much more details. Finally a complete verification tool suite, i.e. STATEMATE , from I-Logix Inc., illustrates the whole verification process. These tools are parts of the verification environment of the Embedded System department of the Carl von Ossietzky Universität, Oldenburg.

Chapter 4 includes the most original part of our work. We define here a class of properties which can be verified more efficiently using invariant check [RS99], rather than model checking techniques. Therefore, we firstly characterize a class of properties represented by timed automata that can be verified using this invariant check. We extend algorithmically this class of properties, and define on the LSC level the properties one would be able to check on this improved way. Most of the properties one could want to verify in real (industrial) usage turn out to belong to this class.

Chapter 5 completes the translation chain our initial LSC specification has to undergo in order to be checked. We hence introduce a simple imperative

[1] The one crafted to kill the werewolf of our nightmares.

formalism, SMI, which allows us to translate the automaton of chapter 1 into a finite state machine understandable by the VIS model checker. This translation allows us, it is worth to mention, to take advantage of the improved verification we described in the previous chapter.

Chapter 6 presents the improvements that were brought to our verification environment by the prototypical implementation of the verification tool chain we described.

Chapter 1

Specification basics

1.1 Introduction to specification

We use many different formalisms as they are all adequate for an aspect of the reality we want to describe, and the vocabulary of the physician is (hopefully) not the one of the mechanician. Besides their differences regarding their field of application, formalisms can also be distinguished with respect to their expressiveness. One will straightforwardly describe some music piece using scores, whereas the same music piece will be written in chords or tabulars to give more room for improvisation. From these different "vocabularies", let us call them *formalisms*, each fits well a different aspect of the same reality, the music piece.

Some formalisms are intuitive, some are not, but the latter could provide a more accurate description. It is obvious that the formalisms we use heavily depend on the habits of our environment. The example of native languages speaks for itself.

We will further use some well-known formalisms in the world of requirements engineering that fit our needs well, trying to describe them on both intuitive and formal way. They were chosen because of historical reasons, or after an in-depth survey of the available languages. The Symbolic Timing Diagrams (STDs) formalism belongs to the first category. It is used to describe the internal behavior of a component. Live Sequence Charts (LSCs) belong to the second, and are used to describe the interactions between many components into a reactive system. LSCs are described in section 1.2. Although STDs are not reviewed in this paper, one can refer to [FJ96] for a description of this formalism. Temporal Logics (described in section 2.2) can also be used to specify interesting properties of reactive systems.

As all these formalisms are used for the same purpose, to specify a requirement, we translate them all into a single formalism, the Timed Büchi Automata. This formalism is expressive and formal enough to allow to reason on it. We describe this intermediate formalism in section 1.3, and show how to translate the previously cited "high-level" specifications languages into one of our automata.

1.2 Live Sequence Charts (LSCs)

1.2.1 The birth of LSCs

In the development of software as well as hardware systems, visual languages are becoming increasingly popular due to their graphical appeal. Especially the telecommunications domain has been using visual languages for many years. In this field the language of *Message Sequence Charts* (MSCs) became a popular means for specifying scenarios that capture information exchange in communication systems [IT96, AE01]. Such languages have been adopted to specify messages passing between components in other fields as well. One token of this expansion is the inclusion of an object-oriented variant of Message Sequence Charts, called Sequence Diagrams, in the UML standard [JRB99] used world-wide as reference formalism.

MCS's language is known to suffer a lack of expressiveness [HP98]. Neither does it provide the formal rigor which we feel is needed for sequence charts to be useful for formal utilization. This motivated the introduction of a sequence chart dialect which remedies these shortcomings: *Live Sequence Charts* (LSCs). LSCs were introduced by Werner Damm and David Harel in 1998, their major improvement with respect to MSCs and previous sequence charts is to provide a means to distinguish between *mandatory* and *possible* behaviors [DH98]. This is done by providing the ability to designate most LSC's elements as belonging to either the *hot* or the *cold* category, characterizing respectively the mandatory and provisional behaviors. This bi-modal property is called the *temperature* of an object.

The next sections present the key elements of Live Sequence Charts (LSCs) following the approach of [DH98]. We introduce some of the extensions to this formalism made by its authors in 2001, and some particular features which were first described in [KW01].

1.2.2 Formalism description

The formalism of LSCs, as appeared in [DH98], provides a rich set of features to describe scenarios, from which we will consider a few, focusing on the core concepts. The graphical representation of this language contributes largely to the easy understanding of LSCs specifications, hence we will illustrate many of the concepts within the LSC shown in figure 1.1 on page 13.

The basic idea of LSCs, as we already told, is to allow a distinction between *mandatory* and *possible* behaviors. To do so, most objects used in the language must be declared to belong to one of those two exclusive modes. Graphically speaking, mandatory, called *hot* elements, are depicted in solid lines, and possible ones (*cold*) in dashed lines.

1.2.3 Constructs of the language

Instances

Each instance represents one participant to the scenario the LSC describes. Instances are represented by vertical lines along which the time runs, from top to bottom. The environment is often depicted as an instance, rather than the border of the LSC, like in the MSC formalism. This allows more flexibility regarding the environment, allowing for instance to specify assumptions on its behavior as for any other instance.

We do distinguish between environment and regular (component) instances. A violation of a LSC which result from a wrong behavior of the environment need to be treated differently than those caused by the system. In the former case we exit the LSC without an error and in the latter case consider it as a real specification violation.

There are four instances in our example LSC on page 13, three are components and the latter is the environment.

Locations

Many locations are linked each to one *event* on an instance. An event related with a location occurs *before* any event related with a *lower* location on same instance. This *chronological order* is relevant on a single instance only, we cannot compare occurrence time of locations on different instances on their relative (graphical) position. The first location of an instance is called *initial* and the last *maximal.*

Messages

Messages are sent between the instances. Their sending or receiving are called events. As the emission of a message must occur before its reception, we can deduce some ordering information between locations on different instances.

We consider two kinds of messages: asynchronous and instantaneous ones, while [DH98] distinguished between synchronous and asynchronous communication.

An *instantaneous* message means that the sending event and receiving event happen at the same time. A *synchronous* message means that the sender is blocked until the receiver has completed whatever request the sender has made. Only the sender and receiver are concerned with this *blocking issue*, the other instances may proceed along their own execution thread. Synchronous communication thus entails a notification of its completion. In a LSCs we require the user to make the return message explicit to highlight the fact that such a process consumes time, and to make the formalism more intuitive. *Asynchronous* messages can take a certain time to get from the sender to the receiver, and do not impose the sender to wait for its arrival, like posting a letter.One could simulate the asynchronous message mechanism using many synchronous messages that transit through a "channel" instance.

Messages are represented by arrows, going from the location associated with their emission event, to the location where they are received. Asynchronous messages have an open-ended arrow, instantaneous ones a solid head. Asynchronous messages are to be drawn slanted as time passes while they are on their way. Instantaneous messages are drawn horizontally, showing the simultaneity of their emission and reception. All messages shown in figure 1.1 on page 13 are instantaneous ones.

Temperature

The temperature concept applies to many objects, indicating how to progress along the instances and messages. Labelling a location with the *hot* temperature (solid line draw) involves the chart *must* progress beyond this location. The analogy is that one can not remain forever in a hot location without burning one's feet. The maximal locations must be cold since we cannot oblige an instance to go further after reaching its last location.

Temperature applies to most of the concepts of LSCs including the entire chart, hot charts are called *universal charts* and cold one *existential charts*. We do not consider the existential charts for the moment as they are not handled by the translation algorithm we use [KW01]. Furthermore the universal interpretation seems to be the natural choice for formal specification, focusing on the fact that an entire system fulfills the specification. The graphical distinction between both is shown in the box surrounding the LSC, which is thus dashed for existential and solid for universal charts.

Combining locations, messages and temperature allows us to express all possible communication behaviors in table 1.1.

temperature	hot	cold
locations	instance run must move beyond location	instance run may stay infinitely at location
message	message will be received once sent	message may be lost

Table 1.1: Temperatures for locations and messages

Conditions

Statements about the system state can be expressed using conditions. Conditions are boolean expressions referring to attributes or data items of the involved components (instances), evaluated when all instances concerned reach the location corresponding to the condition entry. They are graphically represented by an elongated hexagon. Instances which are involved in the condition have their instance axis interrupted by the condition, whereas instances axis of components not participating in the condition continue through the condition. In the example LSC components 1 and 2 are involved in condition C2, whereas the environment and component 3 are not, see page 13.

Conditions can be hot or cold. Hot conditions have to hold unless the scenario fails. Cold ones should be met for the scenario to be validated, but expresses, when violated, that we are not considering a scenario we wanted to talk about, thus exiting without errors. This semantics is not conform to the definition of temperature of [DH98], as it is used here to describe liveness rather than progress, but it allows interesting specification when combined with sub-charts, like conditional behaviors or iterations. We won't detail cold locations nor sub-charts, as they are out of our current focus, but one can refer to [DH98] for more details.

Coregions and simultaneous regions

The chronological order between events is induced by the sequence of locations on one instance, messages and conditions ranging over more than one instance, that can be viewed as synchronization points.

There are two possibilities to change the ordering along the instance axis, should this total order be too restrictive: simultaneous regions and coregions.

To have unordered locations of the same instance, one can put them into a coregion. Coregions are drawn by a dashed vertical line, parallel to the whole concerned instance portion. Within a coregion all events become unordered, as for the arrivals of messages $m3$ and $m4$ on page 13.

A simultaneous region states that all events contained in this region must happen at the same time. This feature has been added to the initial LSC formalism in 2001. It allows to specify the simultaneous observation of several events, as one can meet in the STATEMATE environment described in section 3.3. Such a region is graphically expressed through many events occurring on the same instance at the same height (time), like messages emissions and synchronization through a condition. Before this construct was added to the formalism, a LSC scenario could only describe pure interleaved behaviors, where one instance was allowed to progress at a time. Simultaneous regions make the formalism really adapted to describe parallel execution of communicating devices. Simultaneous regions may not appear in coregions, because otherwise they would imply an order (of simultaneity) to some events of the coregion.

Actions

Actions represent internal behavior of an instance and consequently no impact of an action is observable. They are represented by a rectangle, the border of which depends on the action temperature. Actions are treated as mere comments, we won't consider them in the remainder of this report.

Activation conditions and modes

The range of the specification is described by an activation mode. Three different modes, initial, invariant and iterative, allow us to tell whether the scenario should hold for ever, or only once. A mode is coupled with an activation condition, ranging over the state of the systems, i.e. the instances and the environment. Activation modes are explained in section 1.5, let's just say the iterative mode does not belong (yet?) to the LSC formalism of [DH98].

Both activation mode (A.M.) and activation condition (A.C.) are simply written above the upper-left corner of the chart. As it can be seen, the LSC in figure 1.1 is an `INVARIANT` scenario that will thus be activated every time `ActCond` is evaluated to **true**.

Timer, timing annotations

To be able to express properties about real-time systems, as vital as toasters or airplanes autopilot robots, we use timing annotations and timers. Timing annotations depict the (finite) interval of time for a location to be traversed.

They are written using the mathematical notation for an interval besides the concerned location. For instance, in the example LSC on page 13 the possible arrival of message $m6$ should occur at least one time unit, and at last five time units after component2 left the condition C1.

Timing annotation can be added only to hot locations. Such an annotated location has then to be traversed within the interval.

Timers can be viewed as timing annotations ranging over more than two successive locations of an instance. A timer is represented sometimes by an hourglass with the wanted time values, sometimes only by the time values, with a line going to the location corresponding to the initialization of the timer, and another line going to the location where it runs out. Both locations are located on the same instance.

Sub-charts

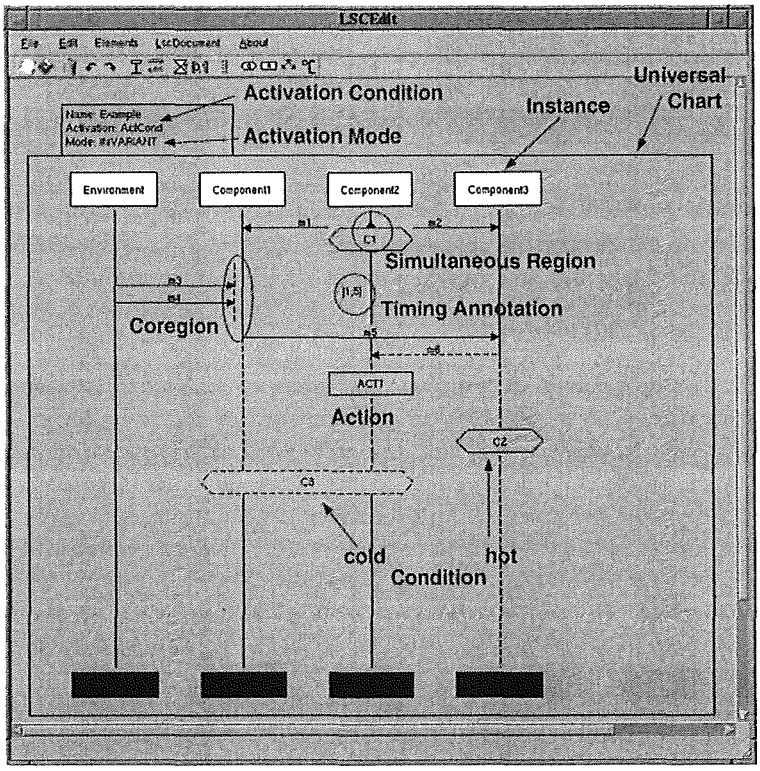
We allow a single use of sub-charts, i.e. LSCs without activation information which are integrated into a main one, as IF-THEN-ELSE construction. Such a sub-chart is characterized by a single condition and two integrated sub-charts, the first of which is activated if the condition holds, otherwise the second one is. This construct differs from the sub-charts of [DH98] inasmuch as different conditions were used in the initial formalism, evaluated at different time, hence allowing both sub-charts to be traversed by the same run of the system. This later behavior seems far from the conduct that one could expect from a real alternative. We thus redefined this construct to fit our needs.

1.2.4 LSC interpretation

We explained how most of the elements of Live Sequence Charts are to be interpreted, at least intuitively, in the previous section. We will now sketch a more formal formalism, i.e. timed automata (1.3), to allow us to describe how the LSC specifications can be translated into these automata (1.4). In the next chapters we describe how these timed automata can be used for formal verification of the initial (LSC) specification. For a more complete syntax and semantics of LSCs the reader could refer to [DH98].

Some restrictions are put in order to simplify the interpretation of LSCs. We do not allow any other element parallel to and independent of a sub-chart. To ensure this we require the sub-chart to cover all instances of the LSC. The second restriction involves the setting of a timer, which has to be bound (via a simultaneous region) to some sort of event. This is adapted to the intuition of a timer, which is set when some event is observed, and then counts time until a subsequent event occurs.

We already made some remarks on the combinations of features: one cannot include a simultaneous region within a coregion, since this would imply an ordering (of simultaneity) on some events of the coregion. The same way we don't allow many conditions to appear in the same coregion, nevertheless many conditions can always be merged in a single one.



Name: Example
Activation: ActCond
Mode: INVARIANT

Figure 1.1: LSC example

1.3 Automata theory on infinite words

LSCs describe the communication behavior of *reactive* systems. Such systems interact with their environment during their all execution, which is often infinite. We want to translate every specification languages used into a single formalism, as stated in 1.1, which should hence accept infinite words, because of the possible endless execution of the system. There exist several different automata which satisfy this requirement. For our purposes Büchi automata on infinite words are sufficient. We introduce them in the remainder of this chapter. In order to be able to treat time aspects, we extend them to timed Büchi automata (TBAs) and finally show how LSCs can be translated into these TBAs.

1.3.1 Finite automata on infinite words

Formal languages are typically characterized as a set of finite words formulated over a finite alphabet [HU77] as are traditional computing languages or human languages, for instance. Such words can be recognized by finite automata and can also be characterized by mathematical regular expressions. The expression $a^*(a|b)$ for instance, describes the finite set of all words beginning with a finite sequence of a's, followed by a single a or a single b.

An automaton is simply a mathematical model of a device that has a constant amount of memory, independent of the size of its input [CGP99]. An automaton on *finite words* can be represented as a graph with labelled transitions, in which the set of nodes are the different possible states of the system and the edges are given by all evolutions possible from any state. Some of the states can be accepting, meaning the system could acceptably stay forever into one of those, without any further evolution. The automaton on finite words in figure 1.2 defines exactly the same language as the regular expression $a * (a|b)$.

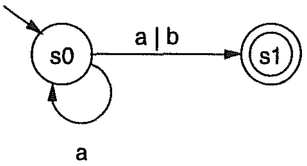


Figure 1.2: The automaton on finite words that accepts $a^*(a|b)$

Languages of *infinite words* can be similarly recognized by finite automata on infinite words, called ω-automata, and are also expressible as ω-regular expressions. The expression $a^*(a|b)^\omega$ for instance, describes the finite set of all words beginning with a finite sequence of a's, followed by an infinite sequence of a and b.

Many different types of finite automata on infinite words have different acceptance conditions, such as Büchi automata, Muller automata or Rabin automata. The reader could refer to [Tho90] or [AD94] for a survey. We will only discuss Büchi 's automata in this report as they are sufficient for our purposes.

Nondeterminism

The automaton of figure 1.2 illustrates a major concern of the automata formalism: when being in state s_0 and next input symbol is a a, should we remain in state s_0 or go to s_1, as both are possible according to the transition label.

This possible choice is called *nondeterminism* and simply means that such nondeterministic automata can have multiple same labels on the outgoing transitions of a same state.

Any nondeterministic automaton on finite words can be translated into a deterministic one that accepts the same language" [CGP99], while this is not the case for Büchi automata.

Formal definition

Formally a (non-deterministic) Büchi automaton (on infinite words) $\mathcal{A}$ is a tuple

$$\mathcal{A} = \langle \Sigma, S, S_0, \longrightarrow, F \rangle, \text{where}$$

- Σ is the finite *alphabet*.
- S is the finite set of *states*.
- $\longrightarrow \subseteq S \times \Sigma \times S$ is the *transition relation*.
- $S_0 \subseteq S$ is the set of *initial states*.
- $F \subseteq S$ is the set of *accepting states*.

A transition $(s, \sigma_i, s') \in \longrightarrow$ represents the change from state s to state s' on input symbol σ_i. We typically write a transition in the form $s \xrightarrow{\sigma_i} s'$.

As we told, an automaton can be represented as a graph with labelled transitions, its set of nodes is S and the edges are given by $\longrightarrow$. In the example shown in figure 1.3 we can find that $\Sigma = \{a, b\}$, $S = \{s_0, s_1\}$, $S_0 = \{s_0\}$ and $F = \{s_1\}$. Initial states are shown with an incoming array whereas accepting states are double circled.

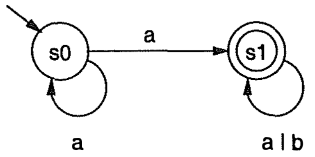


Figure 1.3: The automaton on infinite words that accepts $a^*(a|b)^\omega$

Let $\sigma = \sigma_0\,\sigma_1 \ldots$ be an infinite word over the alphabet Σ. A *run* ρ of the automaton $\mathcal{A}$ over this word σ is defined as a sequence of states: $\rho = s_0\ s_1\ \ldots$, such that

- s_0 is an initial state: $s_0 \in S_0$.
- the target state of each transition is the source state of the following transition: $\forall i \geq 0 : (s_i, \sigma_i, s_{i+1}) \in \longrightarrow$, where many transitions going out from the same state can be labelled by the same character, as the automaton is nondeterministic.

The run ρ over σ can also be written $\rho : s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} \ldots$

A run ρ of $\mathcal{A}$ over σ is *accepting* if some accepting state appears infinitely often in ρ. This criterion is called *Büchi acceptance criterion.*

Formally the Büchi acceptance criterion states that those runs are accepted by a Büchi automaton $\mathcal{A} = \langle \Sigma, S, S_0, \longrightarrow, F \rangle$, which visit some state $s \in F$ infinitely often. Let $inf(\rho) := \{s \in S \mid \forall\, i : \ s = s_i \in \rho \ : \exists\, j > i : s = s_j \in \rho\}$ denote the set of states which are visited infinitely often by a run ρ. A word $\sigma = \sigma_0\ \sigma_1 \ldots$ is then accepted by $\mathcal{A}$ iff there is a run ρ over σ such that $inf(\rho) \cap F \neq \emptyset$. Such a run is called an *accepting run.*

Let Σ^ω be the set of all infinite words over Σ, the language $L(\mathcal{A})$ accepted by $\mathcal{A}$ as expected consists of those words $\sigma \in \Sigma^\omega$, for which there is an accepting run, i.e. $L(\mathcal{A}) := \{\sigma \in \Sigma^\omega \mid \exists\, \rho : s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} \ldots : inf(\rho) \cap F \neq \emptyset\}$.

The language accepted by a Büchi automaton can also be characterized by an ω-regular expression. A language is called *ω-regular* iff it is accepted by some Büchi automaton. The automaton shown in figure 1.3 accepts the language $a^*(a|b)^\omega$, where ω indicates infinite repetition.

1.3.2 Timed finite automata on infinite words

Until now our automata allow us to express properties concerning the sequencing of events (states) of a system. (Un)fortunately many crucial systems depend on *real-time* considerations, remember the toaster, and not only on their qualitative sequence. Rajeev Alur and David Dill [AD94, Alu97] developed a theory on *timed* finite automata on continuous time model. We recall some of their intuition, but rather develop a discrete time framework, since such a context is more intuitive to the system designer and easier to check formally. Follow the way of [AD94] we associate an occurrence time to each symbol of a word, yielding timed words.

Intuitive timing

In the untimed case the behavior of an automaton depends only on the input symbols, i.e., being in some state, the next state(s) of an automaton is (are) determined by the current input symbol. In order for an automaton to accept timed words it needs a means to count time, since the choice of the next state(s) should also depend on the occurrence time of input symbols. Time is introduced into an automaton by adding a finite set of *clocks.* Time passes only when a transition is taken[1]. A clock can be reset to 0 along any transition and at any time the reading of a clock corresponds to the time elapsed since its last reset. With each transition we associate clock constraints, and require the current clock values to satisfy this constraint for the transition to be enabled. With each state we associate a clock constraint called its *invariant* or *stable condition,* and require the time can elapse within a state, i.e., clock values can increase, as long as its stable condition is satisfied. In that way we force the input word to conform to certain timing requirements.

[1] in this way we distinguish from the timed automata of [AD94] where transitions take no time and time only passes within a state

Figure 1.4 shows an example of a timed Büchi automaton $\mathcal{B}$ which has an alphabet $\Sigma = \{a, b, c\}$, a set of states $S = \{s_0, s_1, s_2\}$, a single initial state $S_0 = \{s_0\}$, a single clock $C = \{x\}$, a single accepting state $F = \{s_1\}$, and whose transition relation $\longrightarrow$ and stable conditions are drawn.

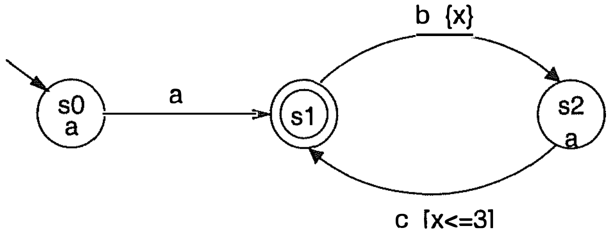


Figure 1.4: timed automaton example

Time interpretation

Time is represented by a sequence of time values which has to satisfy two intuitive constraints: time only advances and time never stands still. More formally, a *time sequence* $\tau = \tau_0,\ \tau_1,\ \tau_2, \ldots$ is an infinite sequence of time values $\tau_i \in \mathbb{N}$ the set of positive integers, for which the following holds:

1. We begin the observation at first available instant: $\tau_0 = 0$
2. Time is strictly increasing: $\forall i \geq 0 : \tau_i < \tau_{i+1}$
3. Time is infinite: $\forall t \in \mathbb{N} :\ \exists i > 0 : \tau_i > t$

A *timed word* over an alphabet Σ is then defined as a pair (σ, τ), where $\sigma = \sigma_0\ \sigma_1 \ldots$ is an infinite word and $\tau = \tau_0\ \tau_1 \ldots$ is a time sequence. The time value τ_i denotes the occurrence time of input symbol σ_i.

Before giving the formal definition of a timed automaton it is necessary to explain what type of clock constraints are allowed as stable conditions and enabling conditions, and how the value of any clock is determined. For our purposes it is sufficient to allow comparison of clock values to constants and conjunctions, even if [AD94] used many more constructors within their clock constraints. Any value of $\mathbb{N}$ can be used as a time constant. The formal definition of the set $\Phi(C)$ of *clock constraints* γ over the set C of clock variables is defined by the grammar:

$$\gamma := x \leq c \mid x \geq c \mid \gamma_1 \wedge \gamma_2,$$

where x is a clock in C and c is a constant in $\mathbb{R}^+$.

In order to evaluate the clocks a *clock interpretation* is needed. A clock interpretation ν assigns to each clock $x \in C$ a value of the time domain. Formally a clock interpretation is a mapping $\nu : C \longrightarrow \mathbb{N}$. Let I be the set of all clock interpretations. The truth value of a clock constraint γ is then given by substituting all clocks in γ by their interpretation:

$$[\![.]\!] : \Phi(C) \times I \longrightarrow \mathbb{B}$$

$$\begin{aligned} [\![x \leq c]\!](\nu) &:= \nu(x) \leq c \\ [\![x \geq c]\!](\nu) &:= \nu(x) \geq c \\ [\![\gamma_1 \wedge \gamma_2]\!](\nu) &:= [\![\gamma_1]\!](\nu) \wedge [\![\gamma_2]\!](\nu) \end{aligned}$$

We say that a clock interpretation ν for C *satisfies* a clock constraint γ over C iff γ evaluates to **true** according to the values given by ν. Regarding the timed automaton of figure 1.4, we can see a clock interpretation ν that assigns 2 to x would satisfy the clock constraint $x \leq 3$ on transition from s_2 to s_1.

We write $\nu + t$ the clock interpretation which maps every clock x to the value $\nu(x) + t$, and we write $\nu\,[Y := 0]$ the clock interpretation which assigns 0 to each $x \in Y \subseteq X$, and agrees with ν over the rest of the clocks in X.

We define these timed automaton more formally in the next section.

1.3.3 Timed Büchi automaton

Our Büchi automaton use a more concise labelling notation than the one used for untimed automaton. Rather than having a distinct transition labelled with each character a of the alphabet $\Sigma = 2^{AP}$[2], the labels consist of boolean formulas over the atomic propositions of AP. One can compare this comprehensive notation to the one used for the clock predicates $\Phi(C)$. We show easily that this simplified representation is equivalent in expressiveness to the extended one simply by giving a mapping from any boolean formula that appears in our TBA to a single character, and define the set of these character as the alphabet of our automaton. Formally we denote by $\mathbb{B}(AP)$ the set of boolean formulas over the atomic propositions AP.

Formally a (nondeterministic) *timed Büchi automaton* $\mathcal{A}$ is then a tuple

$$\mathcal{A} := \langle AP, S, s_0, C, \longrightarrow, F, SC \rangle, \text{where}$$

- AP is a finite set of atomic propositions
- S is a finite set of states.
- $s_0 \in S$ is the single initial state.
- C is a finite set of clocks.
- $F \subseteq S$ is a finite set of accepting states.
- $SC : S \rightarrow \mathbb{B}(AP) \times \Phi(C)$ is a function that maps each state to its stable condition, ranging over the predicates on atomic propositions and clock constraints.

  A stable condition $SC(s) = (b_s, \gamma_s)$ states that the automate is allowed to remain in state s as long as both the predicate b_s and the clock constraint γ_s are satisfied.

[2] which notation is also used by [AD94]

- $\longrightarrow \subseteq S \times \mathbb{B}(AP) \times 2^C \times \Phi(C) \times S$ is the transition relation with labels given by the more concise formulas (rather than individual characters from $\Sigma = 2^{AP}$).

  A transition $(s_i, b_i, r_i, \gamma_i, s_{i+1}) \in \longrightarrow$ represents the change from state s_i to state s_{i+1} with b_i satisfied. The set $r_i \subseteq C$ indicates which clocks are reset when taking the transition and γ_i is a clock constraint that specifies when the transition is enabled, this constraint is evaluated before the clock resets.

In the automaton of figure 1.4 the transitions are simply labelled by the predicate on the alphabet, the clocks to be reset are indicated within braces and the clock constraint is written between brackets.

Semantics

When dealing with timed automata the runs which are considered have to reflect time as well. A *timed run tr* of a timed automaton $\mathcal{A}$ over a timed word (σ, τ) is an infinite sequence of pairs $(s_0, \nu_0)\ (s_1, \nu_1) \ldots$ where $s_i \in S$ is the i-th state visited by the automaton and $\nu_i \in I$ is the clock interpretation in this state, we write

$$tr : (s_0, \nu_0) \xrightarrow[\tau_0]{\sigma_0} (s_1, \nu_1) \xrightarrow[\tau_1]{\sigma_1} (s_2, \nu_2) \xrightarrow[\tau_2]{\sigma_2} \ldots, \text{with}$$

- $\forall i \geq 0 : s_i \in S, \nu_i \in I$.
- $s_0 \in S_0$.
- $\forall x \in C : \nu_0(x) = 0$, all clocks are initialized to zero.
- $\forall i \geq 0$ either $\exists\ (s_i, b_i, r_i, \gamma_i, s_{i+1}) \in\!\longrightarrow\ : [\![b_i]\!](\sigma_i) =$ **true** and $[\![\gamma_i]\!](\nu_i) =$ **true** and $\nu_{i+1} = (\nu_i + \tau_i)[r_i := 0]$
  or $s_i = s_{i+1}$ and, if we call $SC(s_i) = (b_i, \gamma_i)$ we have $[\![b_i]\!](\sigma_i) =$ **true** and $[\![\gamma_i]\!](\nu_i) =$ **true** and $\nu_{i+1} = (\nu_i + \tau_i)$
  The taken transition or stable condition respects its atomic proposition and clock predicates, resets all appropriate clocks and the next clock interpretation is coherent with elapsed time.

To define the run we used the same "interpretation" notation for the formulas on atomic propositions than the one we defined for clocks, except the context of interpretation is here given by the considered input.

The question of which (timed) runs are accepting ones leads to the definition of acceptance criteria for timed automata. We do this analogously to the untimed version, using the same acceptance criteria for automata on infinite words which can be applied to both the timed and the untimed versions.

Again we only consider Büchi acceptance here, for other definitions of acceptance criteria see [AD94]. Timed Büchi automata combine Büchi acceptance with timed automata intuitively described above. As for the untimed case we define the set *inf*(tr) of states of a timed run, which are visited infinitely often:

$$inf(tr) := \{s \in S \mid \forall\ i\ with\ s = s_i, (s_i, \nu_i) \in tr\ : \exists\, j > i : s = s_j,\ (s_j, \nu_j) \in tr\}.$$

A timed word $(\sigma, \tau) = (\sigma_0, \tau_0)\,(\sigma_1, \tau_1)\ldots$ is then accepted by $\mathcal{A}$ iff $inf(tr) \cap F \neq \emptyset$ holds for the corresponding timed run tr. The language accepted by a timed Büchi automaton is correspondingly defined as

$$L(\mathcal{A}) := \{(\sigma, \tau) \in \Sigma^\omega \times \mathbb{N}^\omega \mid \exists\, tr = (s_0, \nu_0) \xrightarrow[\tau_0]{\sigma_0} (s_1, \nu_1) \xrightarrow[\tau_1]{\sigma_1} \ldots : inf(tr) \cap F \neq \emptyset\}.$$

As an example the automaton $\mathcal{B}$ in figure 1.4 accepts the language $a^*(ba^*c)^\omega$ restricted to the words in which a c occurs before the 4-th time unit after a b occurred:

$$L(\mathcal{B}) = \{(\sigma, \tau) \mid \sigma \in a^*(ba^*c)^\omega \wedge \forall i\ \exists j > i : \sigma_i = b \implies \sigma_j = c \wedge \tau_j \leq \tau_i + 3\}.$$

Additional definitions

We finally define an *activated TBA* as a tuple $\langle mode, actCond(\mathcal{A}), \mathcal{A}\rangle$ where *mode* is either `initial`, `invariant` or `iterative`, $actCond(\mathcal{A})$ is the activation condition activating $\mathcal{A}$ and $\mathcal{A}$ is the so-called main automaton, a TBA. Note that the activation condition could be expressed as an automaton.

By convention we can use $\mathcal{A}$ to designate both the TBA and the activated TBA, as it is clear from the context which we are talking about.

We finally define a *TBA specification* as a finite set of activated TBAs: $TBA_{spec} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k\}$

Semantical remarks

The transition relation $\longrightarrow$ does not include self-loops, thus for all $i \geq 0$ such that $(s_i, b_i, r_i, \gamma_i, s_{i+1}) \in\longrightarrow$ we have $s_i \neq s_{i+1}$ (destination node is different from source node).

More practical restrictions are brought to this formalism at the end of the chapter, resulting in a more efficient language. These restrictions are motivated by the particular result of the translation of specifications into TBA, therefore we detail first this translation, called unwinding, in section 1.4, and give more details in section 1.6 about the TBA formalism which has been described here.

1.4 Unwinding LSCs into TBAs

The translation of a Live Sequence Chart (LSC) into a Timed Büchi Automaton (TBA) is called *unwinding*, as we will see the LSC form is (a bit) more compact. The algorithm we explain here has been first described by Jochen Klose and Hartmut Wittke in [KW01], and is based on the procedure of [FJ96] to unwind Symbolic Timing Diagrams.

1.4.1 Intuitive procedure

The purpose of the unwinding procedure is to, finally, get a timed automaton in which each (reachable) state represents one possible state of the LSC, having all the possible states of the LSC included into the automaton. Both the TBA and the LSC should specify the same behaviors, of course.

The possible "states" of a LSC are called *cuts*, they can be viewed as a (curve) line through the chart, across (cutting) all the instances, meeting exactly one location of each of them. Two special cuts are defined, one including all the initial locations, the other all maximal locations. The unwound corresponding states are the (only) initial state and an accepting state of the automaton, labelled by a true stable condition since we are allowed to stay forever in this state.

We begin the unwinding procedure with the initial state, at the top of the chart, and let a "front" down, accordingly to the meaning of each object encountered, every event crossed implies to create a new state in the automaton. We let the cut go downwards until we reach the maximal location on all instances, which state we declare to be acceptant.

1.4.2 Pitfalls to the intuition

The application of this intuitive idea could be straightforward implemented, but we will first clearly define some more critical concepts of the LSCs, such as coregions, simultaneous regions and IF-THEN-ELSE sub-charts. This is done in the next section. The figure 1.5 shows a simple LSC on the left side, with all possible cuts drawn. We can see one critical position is the coregion on component `C2`, where `Msg3` and `Msg4` are received: many cuts go through this position, none should be forgotten by the unwinding procedure!

More formally

We establish a total order on all interesting events of a single instance axis: sending a message, receiving a message, the valuation of a condition, setting a timer, expiration of a timer or the reset of a timer, considering timing annotations are associated to timers. This total order is based on the graphical *position* of the event [KW01], let's call it $position : Events \rightarrow Position$.

All events belonging to the same simultaneous region have the same position, as well as events from the same coregion. The positions of these regions are defined as well, thus extending the definition of $position : Region \rightarrow Position$, where *Region* denotes the set of all the interesting place to characterize a LSC: *Region* = {*Events, Simultaneous regions, Coregions, Initials locations, Maximal locations*}.

We can define the set of *strict predecessors* of a region using this total order along a single instance axis. This set is empty if r is an initial location, and is the set of direct predecessors of r otherwise. Usually $predecessors(r)$ will contain a single event, except if a coregion or a simultaneous region is the strict predecessor of r.

Shared conditions and instantaneous messages force some regions to be simultaneous on different instances. We say these regions belongs to the same *simultaneity class.*

Using the simultaneity classes and the predecessor relation we are able to define the set of *prerequisites* of any region r, the set of regions of the LSC that must be traversed before r can be traversed. $prerequisites(r)$ is obviously empty if r is an initial location, otherwise $prerequisites(r)$ is the set of regions

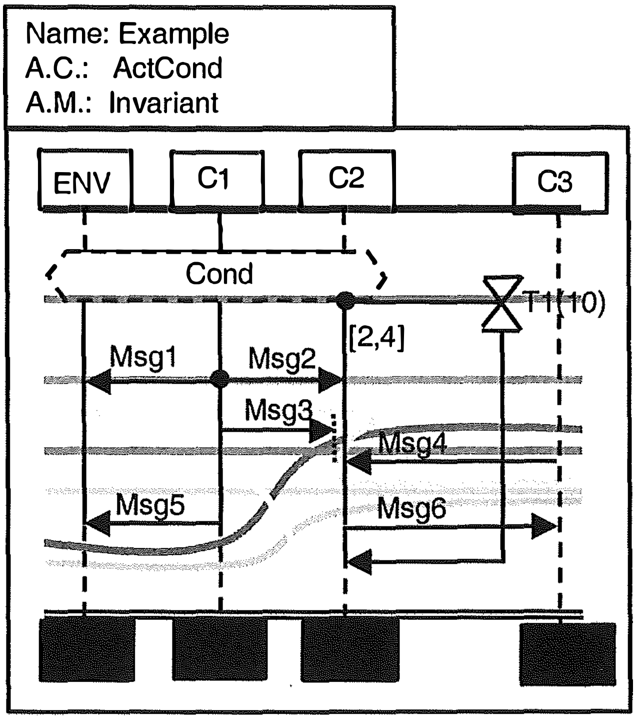


Figure 1.5: Cuts of the unwinding procedure

which are the predecessor of any regions belonging to the simultaneity class of r.

With this definition we are able to formally unwind a LSC by first constructing the simultaneity classes, and then unwinding the regions in such an order their prerequisites have all been unwound before they are. The translations of the resulting automaton corresponds to the successor relation for cuts, i.e the message events that must be fulfilled to get to the next cut.

The unwinding structure obtained from the application of this procedure to the LSC of figure 1.5 is illustrated in figure 1.6

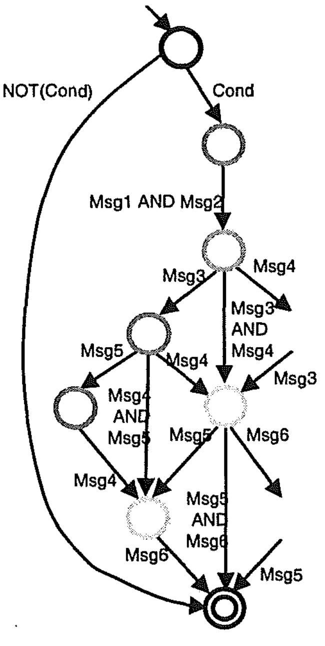


Figure 1.6: Unwinding structure

Considering timing

The depicted unwinding structure cannot express time, we will thus transform it into a Timed Büchi Automaton (TBA). This formalism, described in section 1.3.2, handles timing through clocks, that can be reset and used in *clock constraints*, either on the transitions labels or within the invariants of the states.

As we told in section 1.2.3, timing annotations can only be added to hot locations. They specify an interval of time $[n, m]$, with $n \leq m$, both positive integers. This interval means that the annotated location has to be traversed (left) at least n and at most m steps after is has been reached. Let us imagine a hot location l on instance i is annotated with the time interval $[1, 4]$. This means that when a run reaches l on i, l should remain the active location of i for at least one step, i.e. one clock increment, since we consider discrete time.

Furthermore l has to be left at most at the fourth step since it has been activate, otherwise the run is rejected and an error is generated.

To add this timing information to the unwinding structure we simply consider every hot locations as constrained by a (different) timer (clock). The corresponding timer is reset when the location is reached and a boolean expression constrains the clock value to be within legal range when it is traversed. This boolean expression is simply **true** if there was no timing annotation to this location, otherwise it recalls the timing.

The resulting TBA has *self-loops* on each state, labelled with the condition which has to hold for the TBA to stay in the associated state. These loops are needed since, in our formalism, time only passes when transitions are taken, they are what we called *stable conditions* in the TBA dialect (see section 1.3.2). Its transitions are the conjunction of the predicate from the unwinding structure and adding timing constraints.

Determinism in the TBA

The *activation modes* will be considered later, for the moment it remains as a comment, added to the TBA. A more worrying topic concerns the question of the determinism in the TBA, as it is directly related to the interpretation we give to the LSC's elements.

Three different options can be considered regarding the determinism of the unwound automaton, which depends on the labels we give to the self-loops of each state.

We could completely omit any information provided by timing annotations, providing a totally nondeterministic automaton, which is not efficient for verification purposes. A *strict* interpretation, as considered by [DH98], means that each occurrence of a message has to be explicitly noted in the LSC, and no other is allowed. This interpretation may be too strong, as we do not care whether visible messages are emitted at any time, as long as the desired scenario is fulfilled. The third interpretation, called *weak*, forces the TBA to react to the first occurrence of the expected message, but doesn't restrain the same occurrence at any other time. In a word, unexpected events are ignored. To achieve this mode, self-loops are labelled with the negation of the next message(s) of considered state.

An example

A simplification of the TBA produced by adding the timing consideration to the unwinding structure of figure 1.6 is shown in figure 1.7, with the modifications in bold. This one is a simplification inasmuch as states were put together to allow a better readability.

In appendix A we show a small LSC extracted from the specification related to a light-control system used for train rails. The TBA obtained by application of the unwinding procedure of [KW01] is provided as well.

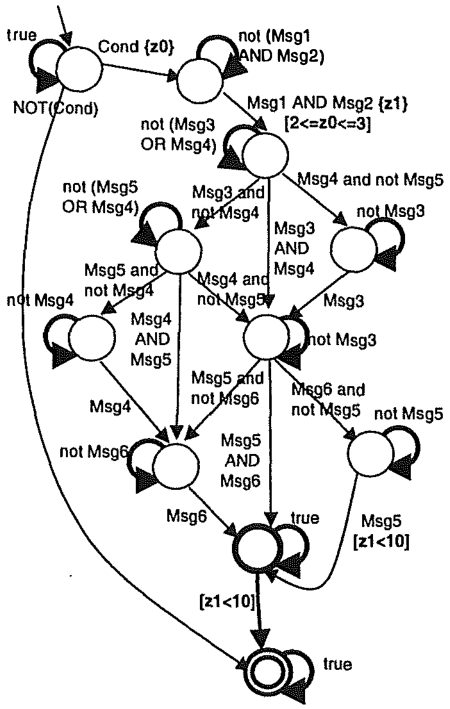


Figure 1.7: TBA resulting from the unwinding procedure

1.5 Activation modes

When describing properties we would like a system to have, it is obvious we also want to tell about the range of this specification: if it holds for ever or only once, for instance. This opportunity is given by adding an *activation mode* to a specification, paired with an activation condition.

Two activation modes are used by [DH98] for the activation of a LSC: *initial* and *invariant*. We add the *iterative* activation mode. These three modes are illustrated in figure 1.8. The horizontal lines symbolize the time, running from left to right, each vertical mark represents an occurrence of the activation condition, which results or not in an activation of the specification, i.e. the verification of the scenario, (in dotted line) according to the activation mode.

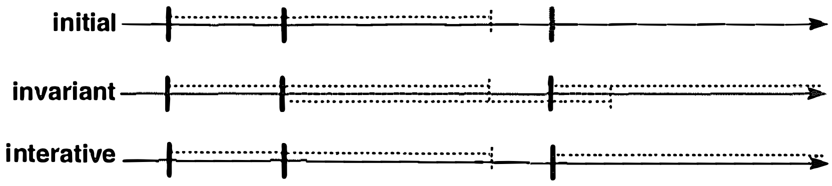


Figure 1.8: Three activation modes

We describe the three activation modes within the next sections and compare their use within both LTL formulas (see 2.2.1) and finite automata (see 1.3.1). Nevertheless the activation mode remain as a comment to the specification, either in LSC or TBA format, we really integrate it into the specification only in a further step of the process, at the SMI level, as explained in section 5.5.2.

1.5.1 Initial mode

The *initial* mode activates the specification immediately, i.e at the first step of the run, the *activation condition* has to hold and the run to be fair to be accepted.

The specification is also accepted if the activation condition does not hold at first step while the *activation exception* does. In this case the specification is not activated, and the run immediately succeeds. If the activation condition is not satisfied at the first step and neither the activation exception, then the run is rejected.

Such a behavior is easily implementable with LTL formulas. If we write *act* the activation condition and P the property coded by the LSC, we can represent the initial mode by the LTL formulae $(act \wedge \bigcirc P) \vee (\neg act \wedge exception)$.

The check of the same condition with a finite automaton is straightforward and results in the automaton which can be seen in figure 1.9.

1.5.2 Invariant mode

In the invariant mode the input sequence is checked *any time* the activation is evaluated to true.

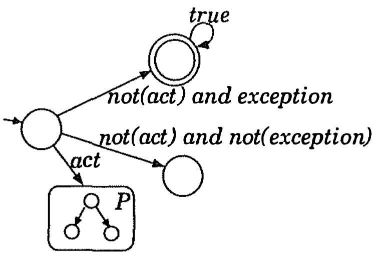


Figure 1.9: Automaton in initial mode

The LTL formula for such an activation mode is of the form $\Box\ (act \implies \bigcirc P)$.

The invariant mode cannot be represented graphically, as we should duplicate all the P evaluation automaton as many time as needed. It is possible to build a *product-automaton* (see section 2.3.2), but this is hard while handling many clocks. Since our automata are complementable one could also check the formula $\neg\Diamond\ (act \wedge \bigcirc\ \neg P) \implies \overline{act \wedge \bigcirc \overline{P}}$, but the double complement can be expensive, as stated in [BH].Finally we could also handle this mode by nondeterministically activate the verification of the specification each time the activation occurs.

Nevertheless we should check only the activations which are able to lead to some problems, as shown in figure 1.10, where the first occurrence (dark vertical bar) of the activation condition doesn't lead to a check of the input sequence as we can forecast there won't be any problem. The second occurrence activates the automaton and provides a witness of a crash (triple vertical bar). We are thus reducing the number of simultaneous checks, this is quite feasible with observers (automata).

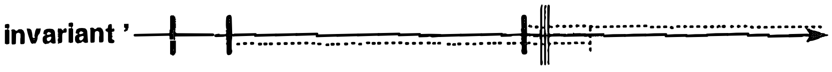


Figure 1.10: Artifact on invariant mode

Instead of trying to determine whether or not we have to activate concurrent checks of the property, we could allow only one instance of this check at a time, but accept more activations if they are not concurrent. This is the purpose of the iterative mode.

1.5.3 Iterative mode

The *iterative* modes sets a lock when an activation occurs. This lock prevents any reactivation of the same specification until it is released, and it is released when we can definitely agree on the specification.

The point is now to define when we are able to agree definitely on the specification, even if this one concerns an infinite word, as it is the case for ω-words. Let's define a *finitely accepting state*, which is a state s such that:

- s is an accepting state.
- s has no outgoing transition, except a unique self loop labelled by true.

We can obviously state that whenever the automaton representing the specification to be verified reaches such a finitely accepting state the requirement is (finitely) fulfilled, recalling therefore Büchi 's acceptance criteria explained in section 1.3.1 which accepts any run as long as it infinitely often goes through an accepting state.

We show in figure 1.11 the finitely accepting states for the LTL formula $p\ U\ q$ with the activation condition r. This automaton is composed of two parts, the first, above the dotted line, represents the activation condition, its state will remain active as long as the activation condition r is not met. This first part *does not* belong to the automaton. The second part is the main automaton, the specification, with 1 fair states (in bold) which fulfills the second property, it is the only finitely accepting state (hence labelled by A).

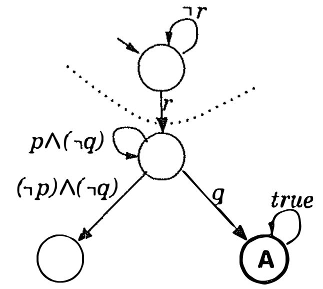


Figure 1.11: Determining finitely accepting states

We want to set and release a *lock* during the check, enabling a new activation only when the current one can't fail anymore.

Some states in which we can be sure of the result, even if the input sequence is infinite, are the finitely accepting states. We should thus release the lock whenever *entering* such a state. We could enhance this definition considering "finitely accepting strongly connected components (SCC)", i.e. SCC with only **true** labels, rather than states, this has not be done yet.

Such a behavior, of releasing the lock when entering any finitely accepting state is simply done through a modification of the transition relation.

The action of releasing the lock actually means to come back from any finitely accepting state either to

- waiting for the activation condition, if is not true yet.
- getting in the initial state of the automaton, if the activation condition already holds.

We show how $\dashrightarrow$ (using the notation described in 1.3.2) has to be modified to exhibit *release lock behavior* in figure 1.12, using the same LTL formula as in the previous example. The added heavy-dotted transition releases the lock, while the **true** self-loop of the finitely accepting state is now labelled be the negation of the activation condition. Such a modification of the transition relation should be done from every finitely accepting states if there are more than one.

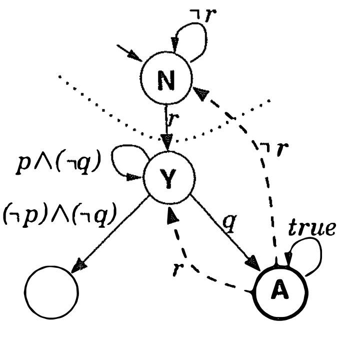


Figure 1.12: Iterative mode lock handling

1.6 Particular TBAs

The Timed Büchi Automata resulting from the unwinding procedure in section 1.4 are a bit particular. In this section we describe some restrictions that can be made on the formalism due to its particular unwound origin.

1.6.1 Activation mode

The activation mode is added to our (unwinding) TBA format as a comment, just as we told. This mere "annotation" will be considered later on, when we effectively check the property, but does not influence the specification at all.

1.6.2 Acyclic Automata

The unwinding procedure builds up the automaton using a total order on the locations, i.e. a location is unwound only when all its prerequisites have already been unwound (we refer to section 1.4.2). This way of doing will always provide us with a TBA that has no back-leading transition, and thus no cycle except self loops.

To highlight the fact that the TBA is cycle-free we remove all self-loops on the states, transforming each of them into an annotation that characterizes its state. This annotation is called *stable condition.*

Its meaning is simply it has to be fulfilled for the automaton to be allowed to remain in the current state. The semantics of a stable condition has been given in 1.3.3

Since the TBAs are acyclic, we are able to define a total order, called *weight*, on all the states of the TBA $\mathcal{A}$: $weight : S \rightarrow \mathbb{N}$, such that $weight(s_i) < weight(s_j)$ if s_i is closer from the initial state of $\mathcal{A}$ than s_j. We define $weight(s) = 0$ if s is the (only) initial state of $\mathcal{A}$ and $weight(s)$ is the sum of the weight of the direct predecessors of s added to the amount of already weighted states else. This weighting can be implemented quite efficiently using a breadth first search procedure with a counter increased by one for every weighted state, i.e. the amount, and hence we state that $\forall i \geq 0\,(s_i, \sigma_i, r_i, \gamma_i, s_{i+1}) \in \longrightarrow : weight(s_i) < weight(s_{i+1})$. This latter assertion states also there are no self-loops within the transition relation.

1.6.3 Remark on clocks

The clocks generated by the unwinding procedures each correspond to a single reference point, either in the LSC or the TBA. Hence, they are always met into the automaton with respect to the same lower and upper bounds. Even if we could find the same clock in many clock predicates of the graph, it will always be used within the same interval. This doesn't change anything to the semantics nor the formalism, but is a simple practical remark.

1.7 To conclude

In this chapter we showed how properties (specifications) of (reactive) systems could be described, and how the LSC specifications of section 1.2 could be unwound (1.4 into Timed Büchi Automata (TBA, 1.3.2). In chapter 2 we explain how formal verification is born, and in the remainder of this report how it can be used to automatically verify the properties we are now able to express.

Chapter 2

Model Checking

2.1 Introduction to formal verification

For as long as programs have existed ones wanted to get rid of their errors. This dream of verification was based on some deeply rooted considerations about the simplicity of program specification, and the idea that verification is always needed.

One can now maintain there are many other ways of obtaining software that is sufficiently reliable for many applications. Simulations are used for a long time to get confidence into any system, implying to run a large number of tests cases through the design. Careful development methodologies [BP94] and well designed testing can give good results in many cases.

On the other hand formal verification uses mathematical techniques to ensure the design conformity, which totally eliminates uncertainty [Wol98].

2.1.1 Like a candle in the dark

The first attempt at proving design correctness relied upon ***invariant proofs*** [Hoa69], but with the limited applicability of this technique and the complexity of proving both inductive steps and termination they were not usable for most people.

Hoare is the first to introduce a formalized programming language, called *Hoare logic* [Hoa69]. He sees a program P as a *transformation* from an initial state to a final state, and thus works with expressions of the form

$$\{\phi\}\ P\ \{\psi\}$$

which means that if ϕ is true before executing P, then ψ is true after its execution.

Hoare then defined some basic program operations including assignments, sequences, alternatives and iterations. Nevertheless, his formalism runs into trouble with more complex constructions like procedures with parameters, pointers, complex data structures or concurrency.

The concurrent composition of programs brought a new challenge to the development of Hoare's logic. Owicki and Gries [GO76] tried to define a concurrent composition rule within the Hoare logic. The concurrent composition of programs P_2 and P_1, written

$$P_1 \mid\mid P_2$$

should fit the requirement

$$\frac{\{\phi_1\}\ P_1\ \{\psi_1\} \qquad \{\phi_2\}\ P_2\ \{\psi_2\}}{\{\phi_1 \wedge \phi_2\}\ P_1 \mid\mid P_2\ \{\psi_1 \wedge \psi_2\}}$$

The difficulty of such a behavior resides in the fact the two concurrent processes can potentially *interact*, with shared variables for example, at any time of their mutual execution. It is thus essential to know what happens *during* the execution, and not only before and after as expressed by this formalism.

2.1.2 The candle becomes lighthouse

Amir Pnueli describes in [Pnu77] *Temporal Logic* as a useful formalism for specifying and verifying correctness of computer programs. This language has become a widely used formalism for reasoning about nonterminating or continuously operating concurrent programs, such as operating systems or network protocols, he calls them "reactive systems".

Temporal logics were first described by Prior, in the fifties, for the perception of time within human languages. Temporal logic is developed by Emerson in [Eme90]. Temporal logic is a modal logic, let us remember such modal logics were initially developed by philosophers to allow expression of possibility. For example, the assertion P may be false in the present world, and yet the assertion *possibly* P may be true if there exists an alternate world where P is true.

Temporal Logic is a particular type of modal logic, allowing to reason about how the truth values of assertions change over time. Typical temporal operators include *sometimes* P which is true now if there is a future moment at which P becomes true and *always* P which is true now if P is true at all future moments.

These ideas were thoroughly explored and Temporal Logic became an active area of research interest. We will explain some useful temporal logics in the next section, but let us first survey the appearance of formal verification.

2.1.3 Automatic formal verification

Through"formal verification" we mean the proof that a system meets a desired property by checking that a mathematical model of the system meets a formal specification that describes the property.

The tools for automatic formal verification are mainly based upon two different theoretical approaches. The first is *temporal logic model checking*, where the properties to be checked are expressed as temporal logic formulas, and the systems are expressed as (in)finite state systems. An example of this approach is the SMV tool, developed at the Carnegie Mellon University, which uses Computational Tree Logic (CTL) model checking to examine whether a finite state system satisfies branching-time temporal CTL formulas. The CTL model checking was first explained by Clarke and Emerson [CGP99]. The formalism as well as the related model checking technique are described further in this chapter.

The second approach called *language containment* makes use of ω-automata to describe both the system and the properties, and verifies correctness by checking that the language of the property contains the language of the system. An application of this approach is the COSPAN tool, of Bell Labs.

Most current tools offer a combination of both approaches, for efficiency reasons, as the HSIS [Bra94] system, from the University of California, Berkeley.

Considering many model-checking tools publicly available, we easily find out that a key question to understand them is the choice of the temporal language they use to specify properties, as this language is one of the primary interfaces of the tool. In the next sections of this chapter we will briefly describe some of the most widely used temporal formalisms and algorithms to perform formal verifications on these formalisms.

2.2 Temporal logics

Let us remember the temporal logics where investigated for describing properties of *sequences of states*, whether finite or infinite. They are an extension of propositional logic, or first-order logic, and use temporal operators to describe temporal (sequencing) properties. Such temporal formulas are given a meaning in a particular state of a sequence, their *interpretation context*. This point of view comes directly from the modal logics where an assertion can be true in some context, while false in another.

One of the major aspect in the design of all temporal languages is their underlying model of time. The nature of time considered induces two different types of temporal logics [CGP99]. In *linear* temporal logics, time is treated as if any moment in time has a unique possible future. Linear temporal formulas hence describe the behavior of a single execution of a program. In *branching* temporal logics each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal logic formulas are interpreted can be viewed as infinite computation trees, each describing the behavior of the possible computations of a nondeterministic program (nondeterminism has been explained in section 1.3.1).

2.2.1 Linear Temporal Logic (LTL)

The logic *LTL* is a linear temporal logic, well described in [CGP99]. Formulas in LTL are constructed from a set AP of atomic propositions using the usual Boolean operators and some temporal operators. These operators give the "context" of interpretation, i.e. the state in which the formula should be evaluated. To understand their meaning let's consider a time-line the unit of which is a single day:

1. X (or $\bigcirc$) is read "next time", it refers to *tomorrow* on our one-day scale.
2. U (or U) is read "until", $\varphi_1 U\ \varphi_2$ means we will meet φ_2 in the future, and until then φ_1 holds.

3. R (or $\widetilde{U}$) is read "releases", it is the dual of "until", this operator is sometimes written V.

Two other operators are then defined as abbreviations:

1. F (or $\Diamond$) is read "eventually", $\Diamond\ \varphi\ = \textbf{true}\ U\ \varphi$.
2. G (or $\Box$) is read "always", $\Box\ \varphi\ = \textbf{false}\ \widetilde{U}\ \varphi$.

The alphabetical notation was first introduced by Prior, the other one comes from Pnueli, they are equivalent and we will use the second one in this report.

Formally, given a set AP, an LTL formula in a positive normal form is defined as follows:

- **true** , **false** , p, or $\neg p$, for $p \in\ AP$.
- $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, where φ_1 and φ_2 are LTL formulas.
- $\bigcirc\ \varphi_1$, $\varphi_1 U\ \varphi_2$, or $\varphi_1 \widetilde{U}\ \varphi_2$, where φ_1 and φ_2 are LTL formulas.

The semantics of LTL is defined with respect to *paths*, or *computations* $\pi = \sigma_0, \sigma_1, \sigma_2, \ldots$, where for every $j \in \mathbb{N}$, σ_j is a subset of AP, denoting the set of atomic propositions that hold in the j's position of π. For a path π, π^i represents its suffix starting at position i, i.e. $\sigma_i, \sigma_{i+1}, \ldots$ of π. We use $\pi \models \varphi$ to indicate that an LTL formula φ holds in the path π. The rules giving the truth of a formula in the first state of a path are the following:

- For all π, we have $\pi \models$ **true** and $\pi \not\models$ **false** .
- For an atomic proposition $p \in AP$, $\pi \models p$ iff $p \in \sigma_0$ and $\pi \not\models p$ iff $p \notin \sigma_0$.
- $\pi \models \varphi_1 \vee \varphi_2$ iff $\pi \models \varphi_1$ or $\pi \models \varphi_2$.
- $\pi \models \varphi_1 \wedge \pi \models \varphi_2$ iff $\pi \models \varphi_1$ and $\pi \models \varphi_2$.
- $\pi \models \bigcirc\ \varphi$ iff $\pi^1 \models \varphi$.
- $\pi \models \varphi_1 U\ \varphi_2$ iff there exists $i \geq 0$ such that $\pi^i \models \varphi_2$ and $\pi^j \models \varphi_1$ for all $0 \leq j < i$.
- $\pi \models \varphi_1 \widetilde{U}\ \varphi_2$ iff for all $i \geq 0$ such that $\pi^i \not\models \varphi_2$, there exists $0 \leq j < i$ such that $\pi^j \models \varphi_1$.

A linear temporal logic formula is a description of a set of infinite sequences, i.e. those that satisfy it. We often interpret those formulas over a *system* with many computations. Formally, a system M is a tuple $\langle AP, S, S_0, R, L\rangle$, where S is the set of states, S_0 the set of initial states, $R \subseteq S \times S$ is a total transition relation (for every $s \in S$, there is at least one s' such that $R(s, s')$), and $L : S \rightarrow 2^{AP}$ maps each state to the set of atomic propositions that hold in it. A computation of M is a sequence of $s_0, s_1, \ldots$ such that $s_0 \in S_0$ and for all $i \geq 0$ there is $R(s_i, s_{i+1})$.

The *model checking problem* for LTL is to determine, given an LTL formula φ and a system M, whether all the computations of M satisfy φ. This problem is known to be PSPACE-complete [SC85], we describe it in section 2.4.4.

2.2.2 Computation Tree Logic (CTL)

The Computation Tree Logic (CTL) is a propositional logic of branching time; i.e., it is based on propositional logic and uses a discrete model of time where, at each instant, time may split into several possible futures [CGP99]. We introduce the syntax and semantics of CTL, less detailed than the LTL ones, relying on the intuition of the reader to make the parallel.

CTL formulas and their Truth semantics

Branching time temporal logic are interpreted over *infinite trees* in which each node is a state, assigning truth values to the atomic propositions

The semantics of a CTL formula is defined over a system $M = \langle AP, S, S_0, R, L\rangle$, where AP is a set of atomic propositions, S is a set of states, $R \subseteq S \times S$ is a total binary relation, S_0 is a set of initial states and $L : S \dashrightarrow 2^{AP}$ maps each state to the set of atomic propositions in AP that are true in that state. R is the next-state relation of the structure. If the system is in state s at a given time instant, it will be in *any* of the successors of s at the following time instant, i.e. the states in the set $\{s' \in S|(s, s') \in R)$. R must be total since CTL formulas have no interpretation for states without successors.

We are now able to define a *path* as an infinite sequence of states $s_0, s_1, \ldots$ such that $s_0 \in S_0$ and $\forall i \geq 0 : (s_i, s_{i+1}) \in R$.

Branching time temporal logic includes two path quantifiers: A for *all paths*, and E for *some paths.*

Let AP be a set of atomic propositions. CTL formulas are defined recursively:

- Every atomic proposition $p \in AP$ is a CTL formula.
- If φ_1 and φ_2 are CTL formulas, then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $A\bigcirc\ \varphi_1$, $E\bigcirc\ \varphi_1$, $A(\varphi_1 U\ \varphi_2)$, and $E(\varphi_1 U\ \varphi_2)$.

Intuitively, $A\bigcirc$ means "all successors", $E\bigcirc$ means "there exists a successor", $A(\varphi_1 U\ \varphi_2)$ means "always φ_1 until φ_2" and and $E(\varphi_1 U\ \varphi_2)$ means "exists φ_1 until φ_2".

Additional temporal operators are defined as abbreviations, in terms of the ones above:

- $A\Diamond\ \varphi \equiv A(\mathbf{true}\ U\ \varphi)$: φ must hold eventually.
- $E\Diamond\ \varphi \equiv E(\mathbf{true}\ U\ \varphi)$: there is a reachable state in which φ holds.
- $E\Box\ \varphi \equiv \neg A\Diamond\ \neg\varphi$: there is some path on which φ always holds.
- $A\Box\ \varphi \equiv \neg E\Diamond\ \neg\varphi$: φ must always hold on all possible paths.

Consider a CTL formula φ and a structure $M = \langle AP, S, s_0, R, L\rangle$, representing the system to be checked. We denote the statement "φ holds in M at state s_0" with $M, s_0 \models \varphi$, using the same notation as for LTL. We write $s_0 \models \varphi$ if the underlying structure M is implicit. $M, S_0 \models \varphi$ to abbreviate $\forall s \in S_0 \subseteq S : M, s \models \varphi$, and $M \models \varphi$ to abbreviate $M, S \models \varphi$. The relation $\models$ defines the formal truth semantics for CTL and is defined recursively as follows:

- $M, s_0 \models p$ iff $p \in L(s_0)$.
- $M, s_0 \models \neg\varphi$ iff not $(M, s_0 \models \varphi)$.
- $M, s_0 \models \varphi_1 \wedge \varphi_2$ iff $(M, s_0 \models \varphi_1)$ and $(M, s_0 \models \varphi_2)$.
- $M, s_0 \models A \bigcirc\ \varphi$ iff $\forall t \in S : (s_0, s') \in R \implies (M, s' \models \varphi)$.
- $M, s_0 \models E \bigcirc\ \varphi$ iff $\exists t \in S : (s_0, s') \in R \wedge (M, s' \models \varphi)$.
- $M, s_0 \models A(\varphi_1 U\ \varphi_2)$ iff for all paths $s_0,\ s_1, \ldots$ there exists $i \geq 0$ such that $(M, s_i \models \varphi_2)$ and $\forall 0 \leq j < i : (M, s_j \models \varphi_1)$.
- $M, s_0 \models E(\varphi_1 U\ \varphi_2)$ iff for some path $s_0,\ s_1, \ldots$ there exists $i \geq 0$ such that $(M, s_i \models \varphi_2)$ and $\forall 0 \leq j < i : (M, s_j \models \varphi_1)$.

Common templates

We summarize the most common CTL templates with the corresponding English language meaning:

1. $A\Box\ p$ is "nothing bad ever happens" ($\neg p$ is bad). Used to specify an invariant, a condition that must be true in all states. Such a formula is helpful for partial correctness (no wrong answers are produced), mutual exclusion (no two processors are in a critical section simultaneously) or deadlock freedom (no deadlock state is reached).

2. $A\Diamond\ A\Box\ p$ is "eventually the system is confined to states where p is always true". It can be used to specify the property of finite number of failures in the system.

3. $A\Box\ (p \rightarrow A\Diamond\ q)$ is "from all reachable states where p is true, something good (namely q) eventually happens". Such formula is used to express total correctness (termination eventually occurs with correct answers), accessibility (eventually a requesting process will enter its critical section) or starvation freedom (eventually service will be granted to a waiting processor). If p is always true, it reduces to $A\Box\ A\Diamond\ q$.

4. $A\Box\ A\Diamond\ q$ is "infinitely often q", i.e., from any reachable state one must reach a state where q is asserted. It can be used, for instance, to enforce a reset condition from any state.

5. $A\Diamond\ q$ is "something good (q) eventually happens" (this one is less restrictive than $A\Box\ A\Diamond\ q$).

6. $A\Box\ E\Diamond\ p$ is "always p possible". It can detect, for instance, the absence of deadlocks, by requiring that it is always possible to reach deadlock-free states. This is an example of a CTL property that cannot be represented by an ω-automaton on words.

7. $A\Box$ **true** forces a complete traversal of the states of the system.

8. $E\Diamond\ p$ is "p is possible". This is another example of a CTL property that cannot be represented by an ω-automaton.

2.2.3 FairCTL

A path is said to be *fair* with respect to a set of fairness constraints if each constraint holds *infinitely often* along the path. Such $\Box \Diamond p$ fairness formulas are expressible in LTL but not in CTL.

To allow to use these really important properties within CTL model checkers too, the model checkers often provide the ability to specify fairness constraints separately from the property.

FairCTL adds to the traditional computational tree logic formula a set of fairness constraints, representing a set of states, each giving a fairness condition. A *fair path* is a path along which each fairness condition is satisfied infinitely often (referring to the Büchi acceptance condition).

FairCTL has the same syntax as CTL, but the semantic is modified so that all path quantifiers only range over fair paths. As an application, let's assume the fairness condition is p, the only paths that will be considered will be those where p is asserted infinitely often. In the literature FairCTL is sometimes written CTL^F.

2.3 Model checking of temporal logic formulas

2.3.1 The choice between linear or branching paradigm

The discussion relative to the merits of linear versus branching temporal paradigm goes back to the eighties [Var98].

The difference in the complexity of linear and branching model checking has been viewed as an argument in favor of the branching paradigm. In particular, the computational advantage of CTL model checking over LTL model checking made CTL a popular choice over the past twenty years, for branching CTL model-checking algorithms run in *linear* time of both the system size and the property length [CGP99] while linear temporal (LTL) model checking took a time exponential within the formula size [WV94]. Even if the theoretical problem of checking LTL formulas is PSPACE-complete, other algorithms provide good results on many typically encountered formulas [GPVW95, SB00], a few of those will be reviewed in the next sections.

On the other hand CTL lacks intuition, and is not usable with compositional verification which is a common way to handle big designs. In contrast, the linear-time way is more intuitive and supports compositional reasoning well [Var01]. LTL is also important because it allows us to express properties, such as fairness, which are not expressible in CTL[1]. A typical fairness property is "if a process is infinitely often executable then it is infinitely often executed", that corresponds to the LTL formula $\Box \Diamond (\textit{executable}) \implies \Box \Diamond (\textit{executed})$.

Because of its good reputation over the past decades, CTL is now served by many efficient algorithms. We can use some of those to verify linear properties following the fact that LTL model checking can be reduced to the language-containment problem, which itself can be reduced to searching for fair paths, which is exactly the FairCTL model checking problem [CGP99]. Such an approach, however, involves the translation of the LTL formula into a transition

[1] we detail in section 2.2.3 how to handle fairness constraints with CTL

system whose size is exponential in the length of the initial formula. As such, it does not enjoy the computational advantage of CTL anymore. Many present researches are concerned in finding a practical way that would enable to use CTL model-checking tools in order to perform efficient model checking on some fragment of LTL [KV98]. [The tool described in chapter 5 allows such a walk-around, providing a means to check efficiently some LTL properties within a CTL-based model checker.]

2.3.2 LTL model checking

The standard approach of LTL model checking [WV94] consists in translating the negation of a given LTL formula into a Büchi automaton and checking the product of the property with the model of the system for language emptiness.

We first show how to translate any given LTL formula into an automaton, without going into too many details as the reader could advantageously refer to the excellent book [CGP99] for more information, which heavily inspired this survey.

The pioneers' way

LTL translation The first way of transforming an LTL formula into an automaton used *Generalized Büchi automata*, which have the particularity to have several accepting sets. The runs over these automata are accepting if they infinitely often satisfy all of these acceptance conditions, i.e. if they infinitely often go through at least one state of each accepting set.

The translation of an LTL formula φ into a Büchi automaton is accomplished by application of the following expansion rules, known as *tableau rules*[CGP99]:

$$\phi_1 U\ \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \bigcirc\ (\phi_1 U\ \phi_2)) \quad \text{and} \quad \phi_1 \tilde{U}\ \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee \bigcirc\ (\phi_1 \tilde{U}\ \phi_2))$$

These rules are applied to φ until the resulting expression is a propositional formula in terms of *elementary sub-formulas* of φ. We call elementary formula any constant, atomic proposition, or formula starting with $\bigcirc$. The expanded formula, put in disjunctive normal form (DNF) is an *initial closure*[2] of φ. The automaton representing the initial LTL formula is then built by the following procedure:

1. Each disjunct of the initial closure identifies a state of the automaton.

2. The atomic propositions and their negations within any term define the label of any incoming transition to the considered state, i.e. the condition that the input word must satisfy in that state.

3. The remaining elementary sub-formulas of the term form the "next part" of the term. They are LTL formulas that identify the obligations that must be fulfilled to obtain an accepting run, hence, they determine the outgoing transitions from the considered state.

[2] also called elementary closure

4. The same expansion process is applied to the "next part" of each state, creating new closures until no new term is produced.

5. The state in the initial closure of φ form the initial states of the automaton.

6. Regarding the accepting conditions we need to impose that for every formula of the form $\bigcirc\ \phi_1 U\ \phi_2$ in the closure of φ, any state that contains that formula is followed by a state that contains ϕ_2. It is sufficient to require that one goes infinitely often either through a state in which $\phi_1 U\ \phi_2$ does not appear, or through a state in which ϕ_2 holds, respectively for every existing "Until" formula.

Such a generalized Büchi automaton can easily be translated into a conventional Büchi automaton expanding the size of the automaton by a factor equals to the number of accepting sets + 1 [CGP99].

Formula negation As we told, for model checking we need an automaton that represents the *bad behaviors*, the ones that are not allowed by the specification. Therefore it is better to first negate the formula, and then translate $\neg\varphi$ into an automaton, as proceeding in reverse order would lead to a double exponential explosion, one to build up the automaton, and a second to complement it.

Product automaton The negated formula automaton (P) is then combined with the system automaton (S) into a so-called *product automaton*, which represents the *synchronization* of the behavior of the system with the behavior of the property. [By synchronization we mean that each time the system automaton executes a transition, the property automaton should also execute one (with the same atomic proposition formula). If no transition is possible for the property it remains in the same state.] The product automaton has typically a state space corresponding to the product of the state spaces of P with S (hence its name), even if some simplifications are possible.

Checking emptiness The next steps of LTL model checking is to check whether the product automaton describes an empty language, if it is the case the property holds along the computations of the system. Checking emptiness of an automaton can be quite efficient and intuitive. Let's recall that infinite accepting runs contain infinitely many accepting states from F, the set of accepting states. Since F is finite we know that any accepting run contains a (finite) suffix such that every state on it appears infinitely many times, thus any state on this suffix is reachable from any other state on it, when such a cycle is *maximal* it is called a *strongly connected component.* Going further we can say that any strongly connected component that is reachable from an initial state and contains at least one accepting state generates an accepting run.

Checking the emptiness of the language of an automaton is therefore reduced to finding a single strongly connected component reachable from an initial state and that contains an accepting state. If there is such a cycle there is an accepting run that can be represented by a ω-regular expression to witness a word accepted by the automaton, whose language is thus not empty.

Tarjan described an algorithm for finding strongly connected components using depth first search, it can be used to decide emptiness of a Büchi automaton in linear time of both the state set size and the transition relation size. More efficient algorithms were described afterwards, most of which used embodies double depth first searches. One can find such an algorithm in [CGP99], among others.

Looking for efficiency

Many improvements have been introduced to this method through the years, which initially gave quite bad results (the process of [WV94] always yields the worst-case result with a number of state exponential in the size of the formula).

A more efficient algorithm has been proposed that works *on-the-fly* [CGP99]. Instead of explicitly extracting a structure that represents all the states of the system, as previously described, this more efficient approach relies upon the automata theory to guide the construction of the system while computing the intersection of the system with the property. This allows to avoid constructing the entire state space of the modeled system in many cases.

Further improvements to the translation of LTL formulas into automata were made using syntactic simplifications [DGV99], and combinations of formula rewriting, boolean optimization techniques and Büchi automaton simplifications [SB00]. The LTL model checking is an active field of research on formal verification. We can hence expect many more interesting results in the coming years, even if interesting results were already found, as explained in section 2.4.4.

2.3.3 CTL model checking

Intuitively the model checking in the branching time framework is the problem of finding the set of states in a state transition graph where the given CTL formula is true.

[CGP99] present a model-checking algorithm for CTL. A CTL formula φ is divided into its sub-formulas $\varphi_1, \varphi_2, \ldots, \varphi_k$ and the states of the state-transition graph associated with a structure $M = \langle AP, S, s_0, R, L\rangle$ are labelled with the sub-formulas that hold for a particular state. Let the size $|\varphi|$ of a formula be one plus the number of its sub-formulas. The algorithm proceeds by successively labelling the states with sub-formulas of size $i = 1, 2, \ldots, |f|$.

Sub-formulas of size one are atomic propositions, i.e. AP, therefore L provides the initial labelling (for $i = 1$). For $i > 1$ and a sub-formula ψ of size $|\psi| = i$, we know that the state graph has already been labelled with the sub-formulas corresponding to the operands of the outermost operator of ψ (because its size is less than i). We can hence determine which states have to be labelled by ψ using the following rules: In the case $\psi = \neg\psi_1$, s is labelled with ψ if s is not labelled with ψ_1, the case $\psi = \psi_1 \wedge \psi_2$ is treated analogously. For $\psi = A \bigcirc \psi_1$ (respectively $\psi = E \bigcirc \psi_1$), s is labelled with ψ if all (some) successors of ψ are labelled with ψ_1.

The case of $\psi = E(\psi_1 U\ \psi_2)$ requires a double computation: first every state that is labelled with ψ_2 is labelled with ψ. Second, any state that is labelled with ψ_1 and has a successor labelled with ψ is labelled with ψ as well. The second step is repeated until no further nodes can be labelled with ψ. This reachability computation can be performed in $O(|S| + |R|)$ time [CGP99]. A similar approach is used for $\psi = A(\psi_1 U \psi_2)$. This yields a CTL model-checking algorithm of time complexity $O\left(|\psi|(|S| + |R|)\right)$.

2.3.4 Language containment

A more expressive logic than CTL (or LTL) would be interesting to model some other properties. One would likely use CTL* [CGP99], to specify, for instance, that q holds "almost always", i.e. always holds after a finite number of transitions ($A\Diamond \Box q$ expresses this, but is not a legal CTL formula). The problem is that such logics have generally more complex model checking algorithms. A feasible alternative is to use another verification paradigm, called *language containment*, based on the theory of ω-automata.

The idea underlying language containment is that a system, represented by a ω-automata S, does satisfy a property, represented by an ω-automata P, iff $L(S) \subseteq L(P)$, where L is the language accepted by the automaton [CGP99].

The common efficient way to check language containment is to compose the given system with the negation of the property and check it for language emptiness, as it is clear that $L(S) \subseteq L(P)$ is equivalent to $L(S) \cap \overline{L(P)} = \emptyset$. The language of the composition is empty if and only if the system satisfies property P. When using this approach of language containment, one must complement the ω-automaton representing the property, and this is hard to do if the automaton is nondeterministic: the new automaton has $O(2^{nlogn})$ states, where n is the number of states of the automaton to be complemented [Tho90, Var01].

2.4 Maturation of model checking

Model checking has evolved from the described algorithms into a tool available on an industrial level. Important milestones on this path were the introduction of symbolic representation of the design under development (*symbolic* model checking)[BCM$^+$90], the capability to focus on components of the design (*compositional model checking*)[CGP99], and finally the use of *abstraction* techniques.

Real applications of model checking requires the ability to deal with large systems, even if the capacities of model checkers are limited by computational limits.

2.4.1 Composition

The *assume-guarantee* paradigm is a well-known means for compositional reasoning [CGP99]. This technique verifies each component process in a isolated way, and then combines the set of assumed and guaranteed properties (hence the paradigm name) in an appropriate manner to establish properties on the entire system resulting from the composition of the components. The major advantage of such a technique is that we never have to compute the whole state-transition graph that represents the design.

Let us assume we have two processes M and M', the respective behaviors of which are *dependent* on each other. The user can specify some assumptions that must be satisfied by M' in order to guarantee the correctness of process M. Let's write it $\langle g \rangle M \langle f \rangle$, where g, f are temporal formulas and M is a process. Since the behavior of M' also depends on the one of M, the user can also express some assumptions regarding M, which must be fulfilled by M in order to guarantee the correctness of process M', for instance $\langle f \rangle M \langle h \rangle$. A typical proof concludes that $\langle g \rangle M || M' \langle h \rangle$ is true, where $M || M'$ represents the system obtained by the composition of both processes.

2.4.2 Abstraction

Abstraction is one technique that closes the gap between the size of the models and the capacity of the checkers by eliminating unnecessary detail from the model.

While compositional reasoning requires the manual activity of decomposing global properties into local components properties, abstraction techniques can be applied to reduce the complexity computationally, i.e. in an automatic way, and hence allow the verification of a design to be fully automatic.

Several techniques of abstraction, like structural, behavioral, temporal or data abstraction have been investigated for long [CGP99]. Some of these require a high degree of user interaction, other ones are completely automatic.

Structural abstraction suppresses details about the implementation's internal structure in the specification, the well-known "black-box" view is such an abstraction.

Behavioral abstraction suppresses details about specified actions that are never activated, i.e. what the component does under operating conditions that never occur.

Data abstraction relates signals in the implementation to signals in the specification when they have different representations. For instance mapping a specified 0 to 100 integer input to a possible output ranging from 0 to 3 within the design allows to save space, since the former values can be reduced to coincide with the second ones.

Lastly, *temporal* abstraction techniques try to match the time units of the specifications with those of the models, possibly resulting in simplifications.

One widely used abstraction technique is based on the concept of *cone of influence* (COI)[CGP99]. This technique attempts to reduce the size of the transition graph by focusing on the variables of the system with respect to a given specification P. This COI contains all parts of the system that are involved in the validity of P, and only those. One can hence prove or refute the property using a simplified design (in the COI), instead of the complete one.

Such abstractions techniques achieve a further reduction of the design complexity, permitting the checking of still larger systems.

2.4.3 Symbolic model checking

Symbolic Model Checking is a particular form of model checking that allows to analyze extremely large finite-state systems by means of symbolic representation techniques (e.g., Binary Decision Diagrams and propositional satisfiability). Symbolic model checking is the core technique for several industrial verification tools and is a main research topic in the area of hardware and software verification.

OBDD formalism

Three independent teams discovered in the late eighties a way to represent transition relations using *ordered binary decision diagram* (OBDDs) [Wol98]. The idea underlying this representation is quite simple: assuming the behavior of a reactive system is determined by n boolean variable v_1, $v_2, \ldots$, v_n we can express the transition relation of this system as a boolean formula $\mathcal{R}(v_1, v_2, \ldots, v_n, v'_1, v'_2, \ldots, v'_n)$ where v_1, $v_2, \ldots$, v_n represents the current state of the transition and v'_1, $v'_2, \ldots$, v'_n represents its target state. This boolean representation can be converted into an OBDD, a full binary tree of depth $2n$ in which

- the leaves are labelled by 0 (false) or 1 (true).
- the interior nodes are labelled by the variables, in a predefined order (hence the name of the representation).
- the outgoing edges of each interior node are labelled by 0 and 1.
- the label of a leaf is the value of the function f for input values corresponding to the labels on the path leading to that leaf.

As an example, let's build the OBDD for the boolean formula $\mathcal{R}(a, b, c) = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$, the corresponding binary decision tree is shown in figure 2.1, where dotted lines represent a value of 0 for the variable labelling the node from which they originate and solid lines represent a value of 1.

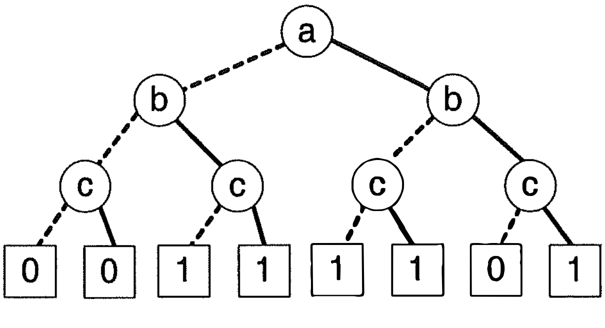


Figure 2.1: The binary decision tree representation of $\mathcal{R}(a, b, c) = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$

A whole technology for efficient manipulation of OBDDs has been developed that works with *reduced OBDDs*. Reduced OBDDs are obtained by eliminating redundant leaf nodes, duplicate interior nodes and redundant tests on the full tree until no further reduction can be made. It has been shown that the maximally reduced OBDD is unique, thus giving rise to a *normal form* for propositional formulas. The OBDD of our sample formula is shown in figure 2.2.

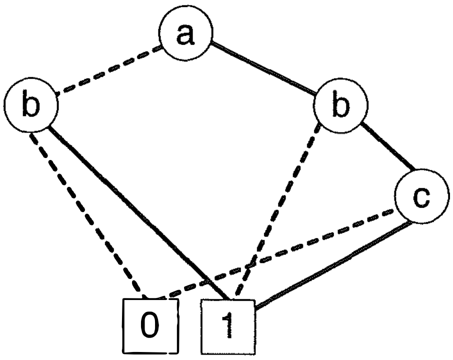


Figure 2.2: The OBDD representation of $\mathcal{R}(a,b,c) = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$

Since the transition relation can be expressed as a boolean formula in two sets of variables, one relative to the current state and the other relative to the next state, this makes it possible to represent predicate transformers and fixpoints as OBDDs.

Model checking using OBDDs

We assume here that the propositions AP are rich enough to distinguish each state, so that a state is uniquely identified by a conjunction of AP formulae. This can always be obtained by increasing AP.

One could, for instance, want to compute the states satisfying the formula $E \bigcirc\ \varphi$. Let $\varphi(x)$ be a boolean formula on the boolean variables x, where x is a vector of boolean state variables, i.e. an OBDD, representing the set of states satisfying φ. The set of states satisfying $E \bigcirc\ \varphi$ can be computed as

$$(E \bigcirc\ \varphi)(x) = \exists x'.(\varphi(x)[x'/x] \wedge \mathcal{R}(x,x'))$$

Where $\varphi(x)[x'/x]$ represents the simultaneous substitution in $\varphi(x)$ of variables x with the corresponding variables x'. This operation, called relational product, can be performed as atomic operation on OBDDs.

These symbolic representation techniques are also called *implicit* representation of the transition system, with respect to the previous approach that *explicitly* generates all possible states.

Propositional satisfiability techniques (SAT)

If we consider *bounded* model checking, i.e. considering only paths of bounded length k, we are able to use more efficient propositional decision procedures [BCC+99].

Bounded model checking is obviously concerned with finding counterexamples of limited length k. This can be computed quite efficiently by incrementing the bound k, and after a predefined number of iterations, state that no counterexample reasonably exists, and that the specification hence holds.

Actually every counter-example (violation) found by such techniques are really counter-example of the specification. But if no counter-example is found for a depth k we cannot conclude with certainty in the correctness of the property, since one could find a violation if the depth is increased. We hence *suppose* that, if the specification holds for k reasonably high, it holds for greater depths too. This *reasonable* limit can be computed, but is exponential for general properties. For safety properties, nevertheless, the number of iterations is bounded by the diameter of the automaton that represents the design.

2.4.4 Efficient LTL model checking

Efficient LTL model checking algorithms are often *linear* within the size of the model, even if exponential in the length of the formula. This high complexity makes real LTL model checking quite expensive for long formulas [CGP99].

Another approach has been used by [EOK94] to reduce LTL model checking to CTL model checking with fairness constraints.

The intuition of their method relies upon an interesting idea: once we have build the *tableau* construction of the negation of the LTL property f to be checked (see section 2.3.2), we can *combine* the design and the property into the input format of a CTL model checker (thus avoiding the expensive product automaton), coupled with *fairness constraints* to make sure that all eventualities of the form $\varphi_1 U \varphi_2$ in f are fulfilled, i.e. whenever $\varphi_1 U \varphi_2$ holds, it has to hold until φ_2 holds. By checking the CTL formula $E\Box$ **true** on the "extended" design, with the described fairness constraints, we can find the set of all the states s such that the formula f holds along every path that begins at s, using a traditional *FairCTL model checking* algorithm. This set of states can finally be translated in terms of the variables of the design, to find out the set of states of the system where the formula f holds.

This approach has been made possible since symbolic, i.e. OBDDs technique are now available to depict large set of states, and in the meanwhile compute efficiently some complex operations (see section 2.4.3).

2.5 Safety properties and invariance checking

2.5.1 Underlying intuition

We presented many formalisms with rather comprehensive expressiveness, LSCs, automata, linear or branching temporal logics. In many applications, however, it is sufficient (and often more intuitive) to specify a system in terms of rather simple assertions.

There are two important classes of properties that one usually wants to verify for a given system:

Safety properties These are properties that intuitively assert that "bad things never happen".

Liveness properties Properties in this class state that "good things happen eventually". They are also referred to as eventuality or progress properties.

One of the simplest forms of temporal specification is in terms of invariants and safety properties. A formula p is an invariant of the system under development if p is true in all reachable states of the design. This can be expressed in CTL, by a *safety* formula of the form $A\Box\ p$, where p is a propositional formula.

Safety properties have been widely researched since they meet the traditional invariant-based proof mechanism: once p is identified as an invariant of the system, one can rely on p to prove other (more complex) properties, using p as induction step: if $M \models (p \implies Prop)$ we can directly state that $M \models A\Box\ Prop$, using a simple induction argument over reachable states.

2.5.2 Checking invariants

The traditional model checking (see 2.3.3), even symbolic, will compute such a safety formula using a fix point characterization, checking whether p holds in current states and $A\Box\ p$ holds in all next states. Such a computation is quite inefficient, and many techniques were made to optimize the particular check of such formulas.

If we already know which states are reachable it is sufficient to perform a test of set inclusion between the set of reachable states (R) and the states represented by p [CGP99], i.e. $R \subseteq P$.

Algorithms were found [RS99] to perform invariant checking on-the-fly. This is especially efficient for falsification tests which does not need to compute the whole reachable states space. This is performed by verifying at each step k of the reachability analysis the following condition:

$$R_k(x) \subseteq p$$

where $R_k(x)$ is the set of states reachable in k or fewer steps. If this test fails, then the invariant is not verified, and a counterexample leading to a state not satisfying the property is provided. We are then allowed not to compute the whole reachable state space.

Chapter 3

Practical model checking

3.1 Model checking tools survey

The actual verification tools follow the path of their ancestors, looking around for means of power and efficiency. The encountered means range from more expressive formalisms to more efficient algorithms. Some approaches that took opposite ways are now often used together, as they provide many interesting, and different advantages. Some of these different approaches were described in section 2.4. Let's remind the techniques of *symbolic* model checking (2.4.3) handles large designs, i.e. systems with many (possibly infinitely many) states, while *composition* techniques (2.4.1) allows to consider designs with many more components

In the remainder of this section we briefly introduce well-known model checking tools, either of academic or commercial origin, which implement the methods presented in the previous sections.

EMC (Extended Model Checker) was one of the first model checkers to be implemented, in 1986. It constructs an explicit representation of the state graph from a program written in a subset of CSP and supports model checking of formulas in CTL with fairness considerations.

Murphy is a description language proposed by David Dill in 1996, the same name has been used to designate an explicit state model-checking system related to this language. Specifications in Murphy are given as simple safety properties, which are verified by explicit state space traversal. Extensions to the Murphy system exploit techniques to reduce the size of the representation of the reachable state space.

SMV was previously developed at Carnegie Mellon University (CMU). It is probably the most widely used symbolic model checker to date, if we associate it with both the academic tools (SMV and NuSMV) and the commercial tool (CadenceSMV). System descriptions in the SMV language are given in a finite state machine fashion, using equations to determine a next-state relation. SMV programs may be structured into parameterized modules. SMV model checks specifications given in CTL with respect to fairness considerations. NuSMV model checker supports LTL specifications, symbolically checked using the approach described in section 2.4.4.

CVE is an industrial verification environment developed at Siemens. It supports model checking of designs described in VHDL or EDIF against specifications given in a temporal logic called CIL, a subset of CTL. If an error is detected, CVE can generate a VHDL test-bench that exposes the fault. Commercial extensions to this product were made that support Verilog as well.

RuleBase, developed at IBM, is an industry-oriented model-checking tool built on SMV that provides a graphical user interface, a temporal logic defined on top of CTL, support for VHDL and Verilog, and debugging support.

VIS integrates model checking with other verification techniques. VIS accepts design descriptions in a synthesizable subset of Verilog, and supports CTL model checking with fairness constraints. Interaction with the SIS synthesis tool is provided through a common intermediate format. The VIS model checker is described in more details in section 3.2.

SPIN is a model checker targeted at the high-level verification of distributed systems. SPIN accepts model descriptions in the model language PROMELA, which provides high-level constructs such as communication channels. SPIN accepts LTL specifications and verifies them using language containment (introduced in section 2.1.3).

The *FormalCheck* tool, marketed by Bell Labs Design Automation, uses COSPAN as verification engine, which is based on language containment.

The *Mocha* specification and verification environment is developed by three American universities. The design is detailed in terms of reactive modules, supporting modular and hierarchical structuring. Mocha recognizes Alternating Temporal Logic specifications (including CTL). Above ATL symbolic model checking based on BDD engines developed at VIS, this tool allows invariant check and decomposition, among other features.

Many other products are available, ranging from a single model checker kernel to a complete tool suite, and more appear each month, since hardware and software verification is a growing area of formal verification.

It is interesting to note that capacity and performance on one side and ease of use on the other, are pointed out by most companies as strong points of their tools. This indicates that these factors are regarded as the most competitive.

In the next sections of this chapter we will describe two real model verification environments, VIS and STATEMATE, which were used together to support the present report, as the second integrates the first. Afterwards we will focus on the common habits and important phases of a verification process.

3.2 The VIS model checker

VIS is a verification and synthesis system for finite-state hardware systems, developed at Berkeley and Boulder. This integrated system allows hierarchical definition, manipulation and verification of designs.

This tool belongs to the hybrid class of model checkers that allows both CTL model checking and language emptiness checks (see 2.1).

3.2.1 VIS overview

As shown in figure 3.1, VIS is composed of three main parts, including a front-end able to read and traverse a hierarchical system described in BLIF-MV, possibly compiled from some high-level language like Verilog; a verification core, concerned with model checking of FairCTL and language emptiness, and finally a synthesis part, using the SIS tool allowing logic optimizations of designs.

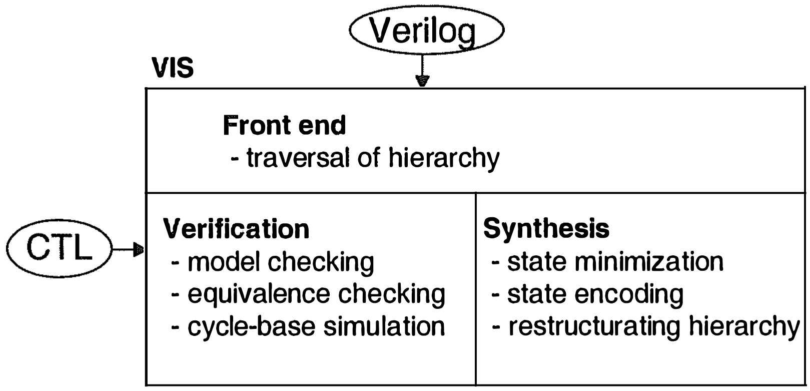


Figure 3.1: VIS overview

The verification core of VIS [BHSV$^+$96] provides tools to check whether models fulfill FairCTL formulas (see 2.2.3) and language emptiness checks, even if language containment is not available yet. This system integrates a support of fairness constraints of generalized Büchi type, i.e. sets of states that must be visited infinitely often.

Regarding the verification facilities, VIS preserves the hierarchical structures within all the operations it provides, as state minimization or symbolic encoding.

3.2.2 Designs description

Specialized languages have been developed for long to describe digital systems, called Hardware Description Languages (HDL). Such systems may be described at a high level of abstraction, such as the *architectural* or *behavioral level*, as well as lower implementation levels, such as the *switch level*, describing the layout of the wires, resistors and transistors on an integrated circuit chip, or at the *gate level* through logical gates and flip flops in a digital system.

A primary use of HDLs is the simulation of designs before the designer must commit to fabrication. The two most widely used languages for digital design are Verilog and VHDL, respectively based on C and ADA. The VIS system currently supports Verilog, even if translators to its intermediate BLIF-MV format could easily be written that accept other HDLs.

Verilog survey

As Verilog HDL [TM91, Hyd95] supports all of the previously described levels, it fits well the possibility of hierarchy support supplied by VIS. This HDL allows mixed-level description of hardware design in terms of static *structures* and dynamic *behaviors.*

A specification in Verilog consists of a set of *modules*. Each of these modules can represent pieces of software or hardware, ranging from simple gates to complete systems such as microprocessors, and it has an interface to other modules to describe how they interact. The *top level module* specifies a closed system containing both test data and hardware models. Component modules normally have input and output *ports* and are driven through *events* which occurs on their input ports, causing changes on the outputs. An *event* can be either a change in the value of a `wire` variable or in the value of a register variable, abbreviated `reg`, or it can be an explicitly generated *abstract event.*

Modules can either be specified behaviorally or structurally, or by a combination of the two. A *behavioral* specification defines the behavior of a digital system (module) using traditional programming language constructs, like IF-THEN-ELSE constructions or assignment statements. A *structural* specification expresses the behavior of a digital system as a hierarchical interconnection of sub modules. At the bottom of the hierarchy the components must be primitives or specified behaviorally. Verilog primitives include logical gates as well as pass transistors (switches).

All the modules of a design run concurrently and communicate with each other through a set of channels, actually wire variables declared in the modules to which these channels belongs. Different access modes are considered depending on the channel type: through wire ports we assume a module can input and output to a channel *instantaneously*, while through a register port it takes one unit time and therefore gets a *storage* element to remember the channel's value.

The top level module invokes instances of other modules. Within each module we can find many legal operations as *continuous assignments* and *procedural blocks* which run in parallel with all module instances. The execution of each continuous assignment, basic block in a procedural block and module instance in one procedural block in the same module is assumed to be atomic within each instant. If there is more than one procedural block in the same module and outputs of one are inputs to another, then the simulated result may depend on how expressions from different blocks are interleaved by the Verilog simulator.

Subset of the Verilog elements supported by VIS

The front-end to Verilog used in VIS, called VL2MV , extracts a set of interacting finite state machines from the Verilog source code into the VIS intermediate format BLIF-MV. Some extensions to Verilog are also supported by BLIF-MV [BHSV$^+$96, Che91], including a particular nondeterministic construction and a way to deal with symbolic variables.

The *assignment* statements are distributed in two categories, continuous and procedural [Hyd95]. Procedural assignments can be either blocking or non-blocking. *Continuous assignments* (written `assign`) drive wire variables and are always active, i.e., they are evaluated and updated instantaneously whenever any input changes. Such assignments describe the combinatorial behavior of a circuit. *Blocking procedural assignments* (written = within a procedural block) acts much like assignments of traditional programming languages: the whole statement is executed before control passes on to the next statement. *Non-blocking procedural assignments* (written <=) evaluates all the right-hand sides for the current time unit and assigns all the left-hand sides simultaneously at the very beginning of next time slot, thus deferring the assignment without blocking the execution of statements in a block. Even if this mechanism is supported in VIS its usage should be avoided since it might introduce unwanted nondeterminism if, for instance, two non-blocking assignments allow different values to the same variable, the final value will then depend on the scheduling of the operations.

A *nondeterministic* construct ($ND) has been added to Verilog [BHSV^{+}96, Che91], allowing to assign nondeterministically many values to a single wire variable, and it is the only legal way to introduce nondeterminism in VIS. For example one can output nondeterministically the values GO or NOGO at a particular *state* using the Verilog code fragment showed in figure 3.2.

```
assign r=$ND{GO,NOGO};
.
always@(posedge clk) begin
.
state = r;
.
end
```

Figure 3.2: Verilog code: a nondeterministic output

VL2MV extends Verilog to allow users to declare *symbolic variables* using an enumerated type mechanism. It can be regarded as declaring a named set consisting of all possible values for the symbolic variable. As an example let us assume a state of a man could be `working, eating, playing or sleeping`, we are then allowed to declare a symbolic type `status_t` which ranges over the possible states of a man in figure 3.3, and further use this type in the declaration of wire or reg variables.

```
typedef enum status_t {working, eating, playing, sleeping};
```

Figure 3.3: Verilog code: symbolic type declaration

Verilog code execution

A Verilog simulator is an event-driven *scheduler*, events generated by modules are scheduled for discrete time and placed on a wait queue. The simulator coordinates between the modules that produce or consume them but it does not generate any event by itself. We call it therefore a *passive* scheduler. The earliest events are at the front of the wait queue and the later events are behind them. The simulator removes all the events for the current simulation time from the queue and processes them. During the processing, more events may be created and placed in the proper place in the queue for later processing. When all the events of the current time have been processed, the next clocking event is chosen by the simulator and simulation time is advanced accordingly to the time stamp of next scheduled event.

The clocking discipline concerning events can either be implicit or explicit[TM91, Hyd95].

When all the transitions of the system are synchronized by an implicit time it is called *implicitly clocked* system. For such systems no hardware resources are to be allocated for synchronization, one just allocates a symbolic latch for each `reg` variable and drops all synchronization variables. This implicitly clocked semantic is default in VIS.

For some designs the operation of a system depends explicitly on several phases of many synchronization signals (*clocks*). The Verilog language provides some types of explicit timing controls over when procedural statements are to occur for such *explicitly clocked* systems, considering that the synchronization has to be completely implemented into hardware. The first type of available control is a *delay control* in which an expression specifies the time duration between initially encountering a statement and when it is actually executed. The second type of timing control is the *event expression*, which allows statement execution. There are also ways to wait for a variable to take some defined value, quite similar to the well-known `wait` statement in C.

3.2.3 BLIF-MV

BLIF-MV is a low-level language designed for describing *hierarchical sequential systems*, it supports nondeterminism [Kuk96]. A BLIF-MV system can be composed of interacting sequential subsystems, each of which can be again described as a collection of communicating subsystems. The original hierarchy specified in BLIF-MV is preserved within the VIS internal data structure, allowing true hierarchical synthesis and verification.

Moreover, this language allows nondeterministic gates, that generate some output from a set of predefined outputs, and hence makes it possible to model nondeterministic systems. Such designs are crucial in formal verification since designs in early stages are likely to contain non-determinism. Lastly, BLIF-MV supports *multi-valued variables*, which can be used to simplify system descriptions.

We can describe the *semantics* of this language as a simple extension of the standard semantic of synchronous single-clocked digital circuit. At every time point, the system is in some state where each latch has a value. An *initial state* of the system is a state where every latch takes a value from its set of initial values. A system can have more than one initial state, in general. At every clock tick, all the latches update their values. These values then propagate through tables until all the wires have a consistent set of values, until *stabilization*. A latch that could be encountered during the propagation, because an output of a table can be an input of a latch, act as a one-time-unit memory, stopping the propagation process on this channel and remembering the channel value until next clock tick. Because of nondeterminism, given a single state, there may be several consistent sets of values.

A design in BLIF-MV is composed of several Finite State Machines (FSM). The interaction of some of these FSMs during the model checking computing is shown in figure 3.4. The different FSMs depicted are

- `IsValidInp Observer` which checks whether the provided inputs are in valid ranges, with respect to their domain.
- `Behavior` which is the FSM representation of the design
- `Properties Observer` which is actually many FSM running in parallel, each representing one specification to be checked.
- `Assumptions Observer` which is the FSM that verifies that the fairness constraints are respected.

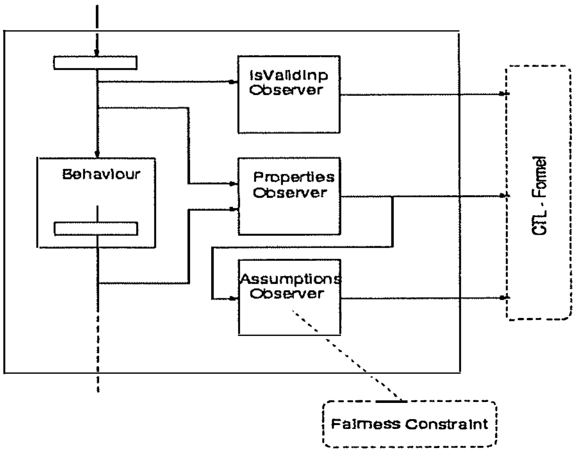


Figure 3.4: The VIS model checker kernel

3.2.4 Language emptiness

We introduced in 2.3.4 the automata-based approach of model checking, that actually checks whether the language of the intersection $L(S) \cap \overline{L(P)} = \emptyset$, where S is the automaton of the system and P the one of the property to be checked.

Vis currently does not support language containment, since the complementation of a nondeterministic FSM is hard (see 2.3.4). But if the user is able to supply the model checker with the complement of the property to be checked, VIS is able to do the emptiness check. Actually this check is reduced to a CTL check of the formula $E\Box$ **true** , which gives an infinite path satisfying the appropriate fairness conditions if there is some.

3.2.5 Safety formulas

The particular CTL formulas of the form $A\Box f$, where f is a quantifier-free formula, are called *safety formulas*, they express that f is true in all reachable states, as we already explained it in section 2.5. VIS implements a specialized (hence expected to be more efficient) algorithm for these formulas, working by forward reachability analysis.

3.3 The Statemate environment

STATEMATE is a widely used graphical specification tool. It is developed by I-Logix Inc., since 1987, and is mainly used for the development of embedded systems [I-L00b].

The STATEMATE tool-set captures the phases of specification, analysis, design and documentation of complex designs. A system under development may be described from three different points of view, covering respectively the structural, functional and behavioral aspects of the design, through different formalisms.

For the verification of designs, STATEMATE uses the technique of model checking through two integrated model checkers. The first one, called SVE, is made by Siemens. It accepts symbolic representation of the system as a finite state machine and (branching time) temporal logic requirements. The second one, the VIS model checker, has been previously described in section 3.2.

A set of tools is included within the tool-set in order to translate the designs into the finite state machines required by both model checkers. In our environment we mainly use Symbolic Timing Diagrams [FJ96] and Live Sequence Charts (section 1.2) for the specification of the expected behaviors.

If a design does not meet the requirement specification the model checker generates a counterexample. This counterexample can be translated by STATEMATE into a timing diagram or a stimulus for the STATEMATE simulator, such a visualization is a convenient help to the designer.

The next sections describe some key features of the STATEMATE verification environment, including some of the supported languages and related tools. We then point out some semantic remarks, more details on semantics can be found in [HP98, HN96].

3.3.1 Features quick tour

We won't describe all the features of this enormous environment, as it is concerned with many related problems, but will focus on the verification aspect with the initial requirements, the specification set that expresses them, help to model the design, verification and correction means, and finally the help provided by STATEMATE for the development of the real system.

Drawing the model

A designer can create a model of a system under development with *Module-Charts* to describe the physical components and their interconnections, *Activity-*

Charts to specify the activities, data and data-control flows between activities. And finally, activities that are not refined into other ones can be depicted using *Statecharts.*

Statecharts can be viewed as finite state machines (FSM) enhanced with hierarchy and other mechanisms like broadcasting between all the charts [HP98]. Many state machines are included into a statechart, possibly activated by one of their active parents FSM. Statecharts describe when and how activities of the design react to stimuli, i.e. they implement the behavior of the specified controller. Timing considerations are added through actions scheduling and events generation, both paired with an occurrence time. Within STATEMATE the real-time behavior of a system is evaluated with respect to a virtual clock.

Data types

STATEMATE supports most of the common data types as bit, integer, real, array, record, union, queue. All the elements of the charts formalisms we cited corresponds to one of theses data-types, and are referenced within three categories that specify how they are handled. *Events* are considered as instantaneous elements, *conditions* are the STATEMATE variant of boolean variables and *data-items* are the memory elements.

Obtaining the right specification

The specifications made may be executed, or graphically simulated, so the system engineer can explore scenarios to determine if the behavior and the interactions between system elements are correct. These scenarios can be captured and included in tests to be run on the embedded system, ensuring that what gets built meets what was specified. All these operations are implemented through different tools within STATEMATE . The executable specification is also an interesting medium between the developer and the end user, confirming the specification meets his requirements.

Statemate as a simple interface for the model checker

All the design information can be automatically translated into the input language of the model checker kernel by the tool. On the other hand the user has to define the properties to be checked, we mostly use STDs or LSCs to describe these properties.

An even more interesting approach of STATEMATE is to help the user to build up the more interesting (often more complicated too) specifications, this is done by the so-called model certifier.

Correction of the design

A *model certifier* is included within STATEMATE that helps the user in defining the properties to be checked. Properties can be expressed by using predefined property patterns. Knowing the semantics of these patterns, the user can sometimes define very complex properties very easily, combining them together and instantiating the pattern parameters with expressions corresponding to his design. Some patterns available in the model certifier are illustrated in appendix B.

The model certifier also allows the definition of very complex assumptions by using the same pattern library. A further extension of this tool is the abstraction capability, which makes it possible to handle relatively large designs for the certification. We described abstraction methods in section 2.4.2.

The errors shown by the model certification phase often provide a *counterexample*, which can be translated into (more) understandable graphical formalisms to allow the user to detect the error within its design. Valuable additional information is provided, relying on all available information, such as the types of the involved variables or counters.

Regarding the properties that can be verified we want to mention some automatically verified *robustness* properties, what STATEMATE calls "debugging" the model. These debugging facilities cover amongst other simultaneous activation of conflicting transitions, several write accesses to a single data-item in the same step and parallel read- and write-access to the same object.

The verification environment offers simple reachability mechanisms to drive the simulation to some user provided state or property, too. One can use such an analysis to verify, for instance, that states indicating fatal errors are not reachable, or to achieve simulation prefixes.

Code generation

STATEMATE allows the user to convert the formal design into C or ADA code (for software developers) and VHDL or Verilog code (for hardware engineers). By creating these virtual prototypes both the developers and the engineers are able to present a prototype to the end-user in the very first stage of the development, ensuring the design is what the user wants.

and many more

Besides the key features highlighted above, STATEMATE provides many more tools, as useful as automatic documentation generator, requirement traceability or revision management interfaces. A more complete documentation can be found at I-Logix.

3.3.2 Semantics remarks

The semantics of STATEMATE , i.e. the one of Statecharts, has been heavily controversed, since this "unofficial" language was used by many, who all gave it their own interpretation, sometimes pretending it was the official semantics of Statecharts, even if there is none.

In the remainder of this section we quickly sketch the way the STATEMATE environment considers time, i.e. the way it should be implemented, more details can be found in [HN96] or in chapter 6 of [HP98].

Time nature

Combinations of the three charts of STATEMATE describe scenarios consisting of steps triggered by external changes and the advance of time. The execution of such a scenario may also generate a chain of steps to be done, so could any internal module.

This illustrates the main concern of the present semantics: the definition of a step, and the way time passes. Time is measured into some unit, common to a whole chart, but different charts can have different time units. A problem is thus to handle time.

Two different time schemes are proposed by [HP98] to face the problem. They only assume no external change occurs within a step, whichever it is, and time only advances at the end of the current step, the execution of which took 0 time unit.

The *synchronous* time scheme assumes the system executes a single step every time unit. Each step of a design, in this semantics, corresponds to exactly one discrete time unit. Time increases uniformly and the environment can influence the valuation of variables between two successive steps. The execution of a scenario using this semantics proceeds in cycles composed of: step execution - time increment - external changes collect - next step execution -...Such a behavior fits well electronic designs for instance, where the real execution is synchronized with a clock signal. This semantics is called *step semantics.*

The *asynchronous* time scheme allows several steps to take place within the same time unit. In general, external changes can occur at any moment between steps, and several such changes can occur simultaneously. This "superstep" performs a chain of internal steps, initiated by an external change, until reaching a stable state, i.e no more internal steps are queued. Then only time advances and the system accepts new stimuli, defining new changes to be queued. This semantics is known as *super-step semantics.*

STATEMATE supports the synchronous (step) semantics, as well as the so-called asynchronous (super-step) semantics.

3.4 Approximations

Many important approximations or language restrictions where made along the history of model checking. Such simplifications were essential to allow formal verification to progress, as they made the faced problems smaller, simply by putting aside some (less interesting) parts of the complete problem.

We describe the approximations we make within our model checking environment in the next sections, regarding our particular TBA format in section 3.4.1, and in the model checking steps we use in section 3.4.2

3.4.1 Specification restrictions

As we previously explained, we do use a set of existing properties (patterns) which are reused to define the specifications to be checked on designs under development. To allow such a modularity we dissociate the *finite* character of a specification which belongs to our library of properties and provide, besides, a means to characterize the "un-finite" character of the property, through the activation modes (see 1.5).

This way of doing is quite intuitive since all *infinite* behaviors of a system that one could want to verify can reasonably be expected to show a cycle, i.e. the same succession of states, which we describe only once, and *reactivate* accordingly to our needs, through the activation modes (see section 1.5).

To express the finite part of the property is one of our real challenges, and this is done using the formalisms we described previously: Live Sequence Charts (see section 1.2), timed automaton (see section 1.3.2), sometimes Linear Temporal Logic formulas (see section 2.2.1) or Symbolic Timing Diagrams (STD). We don't describe the latter formalism in this report as it doesn't provide any new element to the topic covered, one could refer to [FJ96] for such information.

As we are only interested in expressing finite properties we restrain the used formalisms. LSCs (and STDs) intrinsically match our purpose, actually they both use exactly the concept of activation mode and finite property. Therefore no adjustment has to be made.

TBAs resulting from the unwinding procedure have been shown to be acyclic in section 1.6.2, even if we allow to stay in the current state while time goes on. Since this particularity is quite interesting on the efficiency level (as explained in chapter 4), we *impose* it as a formalism restriction.

To be checked, LTL properties have to be translated into automata (see 2.3.2), but the automata formalism we use is cycle-free. This restriction determines some LTL properties we are able to handle, i.e. the ones that guarantee cycle-freedom and finiteness of their corresponding TBA. Formally, the LTL formulas our environment accepts cannot have any *nested* $\Box$ (Globally) operator, as it would be unwound into a cycle.

3.4.2 Environment approximations

Finite state formal verification cannot, at the present time, deal with *infinite data types*. Such types are used to declare design variables, e.g. unbounded integer or reals. It is worth mentioning that the verification tool set of VIS offers abstraction and approximation techniques in order to be able to apply finite state verification methods to these designs. Such techniques were introduced in section 2.4.

We specify our properties using different *linear time* formalisms because such an intellectual process is easier for the designers. The linear time framework is more natural for anyone rather than trying to think "branchingly". The experience of IBM with the RuleBase system gives evidence about the difficulty for most users to understand non trivial CTL formulas, as it is simply much harder to reason about computation trees than about linear properties.

The VIS model checker is a *branching time model checker*, as we presented it in section 3.2. Hence the properties are approximated into their branching counterpart before being checked.

We transform the LTL formula into a CTL property by adding A before any temporal operator, thus checking the target formula on any possible path. It is obvious this changes the meaning of the property, for instance the

LTL equivalence $\bigcirc\ (p \vee q) \equiv \bigcirc\ p \vee \bigcirc\ q$ does not hold anymore in our approximated CTL: $A \bigcirc\ (p \vee q) \not\equiv A \bigcirc\ p \vee A \bigcirc\ q$ which is stronger. When the CTL-approximated model checking fails we are not always able to guess whether it failed because of the models that does not fulfill the requirement, or because of our approximation being too restrictive with respect to the initial specification.

Chapter 4

Finite acceptance on infinite words

The habit in formal verification is to define an interesting class of languages and study it to find its properties. This chapter is written the other way round. We first define interesting properties, and then characterize the language for which these properties hold. Indeed, we had a good intuition from the beginning of the expected result, which turns out to be effectively of interest for model checking.

In section 4.1 we define possibly interesting acceptance criteria and evaluate their strength. The second of these criteria seems to be of interest and we compare it with the acceptance condition of Büchi in a theorem in section 4.2. We extend the range of the theorem (4.2.2) and finally define a class of specifications, in section 4.3, that can be verified efficiently using state reachability techniques instead of traditional (fair) model checking ones. A small conclusion is given in 4.4 that situates our propositions within the current formal verification context.

4.1 Acceptance criteria

4.1.1 Many criteria

We explained in section 1.3.1 the acceptance criterion of Büchi on finite automata running over infinite words: any run ρ of the automaton $\mathcal{A}$ over the word σ is accepting iff some accepting state of $\mathcal{A}$ appears infinitely often in ρ.

We could offer other acceptance criteria, easier to check than the one of Büchi . A property expressed by a nondeterministic TBA takes at least a time $O(2^{n \ log \ n})$ to be checked, where n is the size of the set of states (see 2.3.4). Many other acceptance conditions are described in [Tho90], such as the *1-acceptance* which accepts a word $\alpha \in \Sigma^\omega$ iff α belongs to an open ω-language.

Finite acceptance

A possibly interesting new acceptance criterion relies on the definition of a *finitely accepting state*, defined in section 1.5.3. Such a state is an accepting state with no outgoing transition, and its stable condition is simply **true**.

We say that an infinite run ρ on a finite automaton $\mathcal{A}$ is finitely accepted *iff ρ reaches a finitely accepting state of $\mathcal{A}$.*

More formally let $\mathcal{A}$ be the TBA $\langle AP, S, s_0, C, \longrightarrow, F, SC\rangle$ and $(\sigma,\tau) = (\sigma_0,\tau_0)\ (\sigma_1,\tau_1)\ldots$ a timed word with $\sigma_i \in AP$. We say that $\mathcal{A}$ *finitely accepts* (σ,τ) iff there exists a run $tr : (s_0,\nu_0) \xrightarrow[\tau_0]{\sigma_0} (s_1,\nu_1) \xrightarrow[\tau_1]{\sigma_1} \ldots$ where, for some $j \geq 0 :\ s_j \in F \wedge \forall (s_k, b_k, r_k, \gamma_k, s_{k+1}) \in \longrightarrow : s_k \neq s_j \wedge SC(s_j) = \mathbf{true}$.

Evaluation

The language defined by the finite acceptance on an automaton $\mathcal{A}$ is clearly a subset of the language defined by the Büchi acceptance condition on $\mathcal{A}$, since any finitely accepting state belongs by definition to F, the set of accepting states of $\mathcal{A}$. Nevertheless this finitely acceptance criterion seems not to be powerful enough to be of any use. For instance, it does not accept a single word on the timed automaton of figure 1.4 on page 17, since there is no finitely accepting state in this automaton.

On the other hand the model checking of a property represented by a Büchi automaton on words is quite hard to compute: it implies an exponential blow up using language containment (see 2.3.4). We describe in section 4.1.2 another acceptance criteria, which turns out to be strong enough to be of interest, especially on some particular TBAs, allowing the represented property to be computed in a time linear of the state space 4.1.3.

4.1.2 Non-failure acceptance

Let us figure a timed ω-automaton such that all its unfair (not accepting) states have to be left within a finite (bounded) time. We say that such a state is *constrained* by a clock, i.e. its stable condition is labelled by a clock predicate which sets an upper bound on (at least) one clock, and this bound is not arbitrary big. The same way, we call an automaton constrained iff all its unfair states are constrained.

Figure 4.1 exhibits such an automaton, all the unfair states (s_0 and s_1) of which are constrained.

Lemma 1 One can assert that, if all the unfair states of an automaton are constrained, any accepting run will finitely reach and stay in an accepting state. This holds because there are no backleading transitions in the automaton. In our example a fair state (s_2 or s_3) is reached within at most 8 steps (2 before reaching s_1 and 6 from s_1 to s_3).

We now introduce a "sink state" in our automaton of figure 4.1 to make it complete. An automaton on Σ is *complete* if it has a run on every word in Σ^ω. The sink state is such that we are *always* able to take at least one transition or stay in the current state, whatever the input may be. It should not change

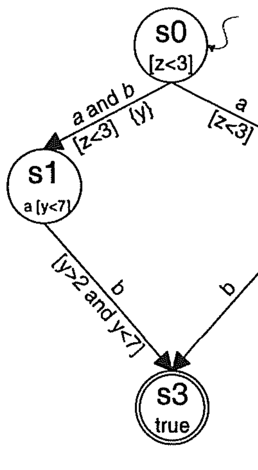


Figure 4.1: A constrained automaton

the language of the automaton and is therefore not accepting. Moreover, it is labelled by a **true** stable condition, has no outgoing transitions and is reachable from any state of the graph through a transition labelled with the negated disjunction of all outgoing transitions and stable condition of the considered state, as illustrated in figure 4.2.

We certify that this added sink-state does not change the language recognized by the Büchi automaton since this state is not accepting and cannot be left once reached. Therefore any run that goes into it could not pass infinitely often into an accepting state and thus not recognize new words, referring to the Büchi acceptance criteria. It is obvious that the addition of states and transitions does not either restrict the language of the modified automaton.

It is now clear that every run of this completed automaton which never goes into the sink state will finitely reach an accepting state (because of lemma 1), and is therefore an accepting run. We notice that this automaton is not constrained anymore, because of the introduction of the sink state, the stable condition of which is **true**. Such an automaton is called a *completed constrained* automaton.

We express the above observations by the *non-failure* acceptance criteria:

Non-failure acceptance criteria *We say that an infinite run ρ on a completed constrained (timed) automaton $\mathcal{A}$ is* non-failure accepted *iff ρ never reaches the failure state of $\mathcal{A}$.*

Formally

To be accurate let us remember all the hypotheses that are made on any completed constrained automaton $\mathcal{A}$:

- $\mathcal{A}$ is complete,

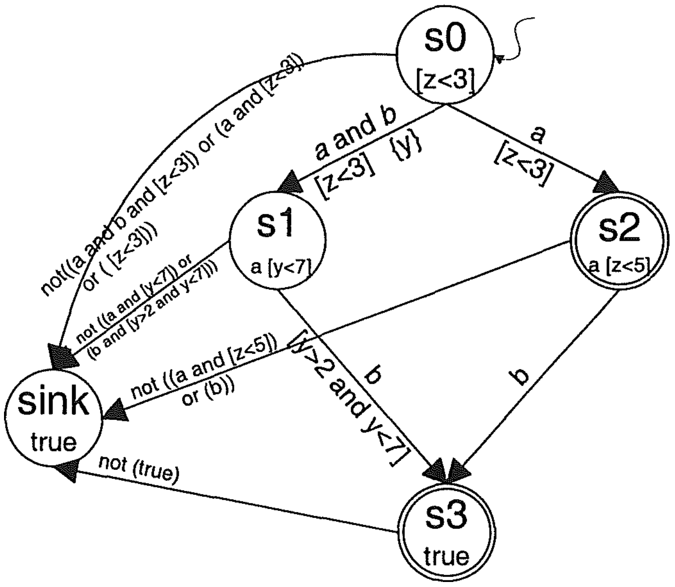


Figure 4.2: A completed automaton

- the transition relation of $\mathcal{A}$ has no backleading transitions, i.e. $\forall(s_i, \sigma_i, r_i, \gamma_i, s_{i+1})$ belonging to its transition relation we have: $weight(s_i) < weight(s_{i+1})$, using the total order on states defined in 1.6.2.
- except its sink state (sometimes called failure state), whose stable condition is **true**, every unfair state of $\mathcal{A}$ is constrained by a clock, i.e. its stable condition defines a not arbitrary big upper-bound on a clock.

Let $\mathcal{A}$ be the TBA $\langle AP, S, s_0, C, \longrightarrow, F, SC\rangle$ and $(\sigma,\tau) = (\sigma_0, \tau_0)\,(\sigma_1, \tau_1)\ldots$ a timed word with $\sigma_i \in AP$. We say that $\mathcal{A}$ non-failure accepts *(σ, τ) iff there exists a run tr : $(s_0, \nu_0) \xrightarrow[\tau_0]{\sigma_0} (s_1, \nu_1) \xrightarrow[\tau_1]{\sigma_1} \ldots$ where, for all $j \geq 0$: $s_j \notin F \implies \exists(s_k, b_k, r_k, \gamma_k, s_{k+1}) \in \longrightarrow$: $s_k = s_j$.*

Notations

The same way a Büchi automaton designates a finite automaton on infinite words accepting runs accordingly to the Büchi acceptance criterion, we define a *NFA automaton* as a finite automaton on infinite words accepting runs accordingly to the non-failure acceptance (NFA) criterion.

4.1.3 Invariant check

The last formulation of the NFA criterion highlights that it is actually a safety condition. It states that "something bad (i.e. reaching the sink state) never happens". As we told in section 2.5 such properties are some of the easiest formulas to be checked by a model checker, since they express an invariant property, and hence reduce to checking the reachability of a state.

Computing the reachability of the sink state can be done on-the-fly, as explained in section 2.5.2. Such a verification is far more efficient than a complete model checking procedure. [A property translated into an NFA automaton can be verified by simply checking $\Box$ ($\neg$*failure*). If we let *failure* represents the fact to enter the sink state. Therefore, we do not need to check a formula with fairness constraints, which would be expressed as $\Box \Diamond$ (*constraint*).] If the same property can be expressed using NFA or Büchi acceptance, one should use the NFA since its verification can be implemented on a more efficient way.

In the next section we describe a class of automata the expressiveness of which remains the same, whether they are considered as NFA automata or Büchi automata.

4.2 Expressiveness theorem

We propose the following theorem:

Expressiveness theorem *Completed constrained (timed) automaton have the same expressiveness whether they are considered as NFA automata or as Büchi automata.*

To prove this theorem we find first that any run accepted by the condition of Büchi condition is also accepted by the NFA criterion. The criterion of Büchi states that any run is accepting iff it passes infinitely often through an accepting state. Because there is no cycle in the transition relation of a completed constrained automaton, the only way to reach infinitely often an accepting states does never passes through the sink state, which cannot be left once reached. Therefore the second criteria holds for any such run.

Secondly, we show that any run accepted by the NFA criterion is accepted in the sense of Büchi too. The NFA states that every accepting (infinite) run never goes into the sink state. We are able to ensure it will finitely reach and stay into an accepting state, thanks to lemma 1 (on page 4.1.2), since any other (unfair) state has to be left finitely. Therefore Büchi 's criterion is satisfied as well. □

The completed constrained automata are only a subset of the complete TBA formalism, since their requirements are severe. We show in the next sections they are nevertheless sufficient to express most the properties one could want to verify.

4.2.1 Transitive clock constraints

Binding clocks and paths

A clock constraint, or clock predicate, is expressed either on a transition or on a stable condition, but it actually constrains a whole run. In fact, an upper-bound on a clock, either labelling a transition or a stable condition, constrains the length of the path portion between this transition and the earlier resets of this clock, or the initial state if there is no reset before it. We say that a state (or a transition) belonging to such a path portion is *transitively constrained* by a clock.

The automaton 1 of figure 4.3 highlights all the transitively constrained transitions and states in dotted lines. We consider there's only one clock c in this automaton. The R tag symbolizes a reset of the clock c, and the U (as "use") a constraining predicate (upper-bound) on c. The number in each state represents its weight, using the weighting function of 1.6.2. Other weighting functions can obviously be used, as long as they define a total order on the states.

Explicit transitive constraints

We propose to constrain (explicitly) any label that is transitively constrained by a clock upper-bound. We consider only upper-bound predicates since our purpose is to broaden the use of NFA automata, defined in 4.1.2. [We could transitively repeat the lower-bounding predicates as well, but choose not to do since the overhead is heavy for only a few interesting results. The lower-bound transitivity is actually almost erased by the stable conditions allowing time to pass while remaining in a state. The reader could nevertheless adapt the considerations on upper-bounds to lower ones.]

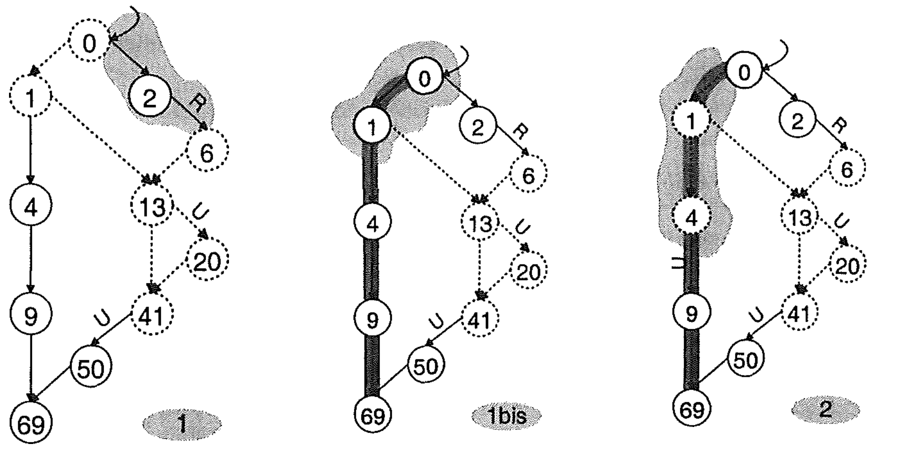


Figure 4.3: The states and transitions which are transitively (1) and globally (1bis, 2) constrained by clocks

A first idea to highlight the transitivity of clock bounds is to constrain (explicitly) any transitively constrained stable condition or transition of the automaton. Therefore, we propagate any clock upper-bound, going backwards from this bound up to a reset of this clock or to the initial state, as shown in the first automaton of figure 4.3.

The constraint on c in the label of the transition from 41 to 50 constrains transitively all the states and transitions leading from 0 to it, except those in grey since a reset is met between 2 an 6, that prevents all previous elements of this path from being constrained.

Some portion of the newly constrained paths also belongs to (other) non-constrained paths as the dark grey path in automaton 1bis. We hence restricted the language the automaton recognizes with these explicit constraints. To preserve the language of the automaton we should constrain a label only when *every* path that includes this label is transitively constrained on the portion including this label. Such a transition (or state) is called *globally constrained.* This is applied in the automata `1bis` and 2 of figure 4.3.

In automaton `1bis` (as in 1), the state weighted 1 cannot be constrained since it belongs to the path in dark grey, in which the state 1 is not transitively constrained by c. In the last automaton (2) the dark grey path sets an upper-bound on c in state 1 as well as any other path that includes this state, it can therefore be explicitly constrained without changing the language of the automaton.

In the next section we give a procedure that handles this clock manipulation.

4.2.2 Clock algorithm

Hypotheses

- We consider the automaton is weighted. This means that we can easily see between two states which is farthest from the initial state. We use the *weight* function described in section 1.6, which assigns to each state the sum of the weight(s) of each of its parent(s)-state(s) plus the amount of already weighted states, and assigns 0 to the initial state.
- There is no transition in the automaton which can be statically evaluated to **false**. This ensures not to constrain too much labels, hence reducing the language of the automaton.

Notations

We explained in the semantics of the TBAs (see 1.6.3) a clock c will always be used with the same lower and upper bounds, let's call the upper bound predicate of these bounds *upbound*(c). If it was not the case, we would have define *upbound*(c) as the predicate defining an upper bound on c which is the supremum of all upper bounds on c in the automaton.

To constrain a label *explicitly* it is sufficient to add this *upbound*(c) predicate to its clock predicate. As a possible optimization the *upbound*(c) added could be decreased by one, as one transition is taken to reach the original bound.

The *range* of a clock predicate is defined as the set of clocks constrained by this predicate, for $p \in \Phi(C)$ it is written $range(p)$. The same way, we define the set of clocks on which the predicate puts an upper bound, for $p \in \Phi(C)$ it is written $uprange(p)$.

$$uprange : \Phi(C) \longrightarrow 2^C$$

$$\begin{aligned} uprange(c \leq b) &= \{c\} \\ uprange(c \geq b) &= \emptyset \\ uprange(p_1 \wedge p_2) &= uprange(p_1) \cup uprange(p_2) \end{aligned}$$

Where c is a clock in C and b is a constant in $\mathbb{N}$. Of course $c \in uprange(upbound(c))$.

We abbreviate the set of transitions which emerge from a state $s \in S$ with $outgoing(s)$, and the set of transitions that reach this state with $incoming(s)$. For the sake of readability we associate the stable conditions to usual transitions, like self-loops on their state, hence the stable condition of s belongs to both $outgoing(s)$ and $incoming(s)$.

The information contained in a label (either of a transition or a stable condition) can be partially accessed using the following predicates: $Reset(t)$ gives the set of clocks which are reset when the transition t is taken, $Cpred(t)$ is the clock predicate of t and $origin(t)$ gives the state from which the transition t emerges, of course $origin(t) = s$ iff $t \in outgoing(s)$.

Finally we define formally the notations we already used. We say a transition t is *constrained by a clock* c if $c \in uprange(Cpred(t))$ and that a state s is constrained by a clock c if its stable condition is. A transition or a state is simply *constrained* if it is constrained by at least one clock. Given a TBA we define the function $constraining\_paths : S \times C \to \mathbb{B}$, such that $constraining\_paths(s, c)$ is true iff for all $t \in outgoing(s)$ we have that c constrains t, so that the clock c constrains any path that includes s. If $constraining\_paths(s, c)$ holds we say that s is *globally constrained* by c.

Clock algorithm version 1

The following procedure, called clock algorithm, constrains explicitly all the transitions and stable conditions that are globally constrained by any clock. Such an automaton is called *globally constrained* after this transformation.

We give this algorithm as if there was only a single clock c in the TBA's clock set, as it is more readable. We can easily extend it to more clocks simply using an adapted data format as the set of clocks is finite, or by rerunning the procedure for any clock $c \in C$ if one does not care for efficiency, which is nevertheless one of our key topic in this report .

```
        $Optimized := \emptyset$
        $GloballyConstrained := \{s \in S | constraining\_paths(s, c)\}$
(inv)   While ($GloballyConstrained \neq \emptyset$) do
            Choose $s' \in GloballyConstrained$ with the maximal weight
    (1)     $\forall\, t \in incoming(s')$
        (2)     if $c \notin Reset(t)$ then
            (3)     $Cpred(t) := Cpred(t) \wedge upbound(c)$
                    if ($constraining\_paths(origin(t), c)$) then
                (4)     $GloballyConstrained := GloballyConstrained \cup \{origin(t)\}$
    (5)     $GloballyConstrained := GloballyConstrained \setminus \{s'\}$
            $Optimized := Optimized \cup \{s'\}$
```

Proof

To prove the automaton recognizes the same language after being globally constrained by the above procedure we propose an invariant on the while loop ("inv" label). A state is called *optimized* if all its incoming transitions are constrained by each clock that constrains this state, in this case by c.

Invariant

- $GloballyConstrained = \{s \in S | constraining\_paths(s,c) \wedge s \notin Optimized\}$, *GloballyConstrained* contains all the states which are globally constrained by c and have not been optimized (yet).
- $Optimized = \{s \in S | constraining\_paths(s,c) \wedge \forall t \in incoming(s) : c \notin Reset(t) : c \in uprange(Cpred(t))\}$, *Optimized* contains all the states which are globally constrained by c and have already been optimized.

This invariant is quite simple to prove: at the first step these two conditions hold, since it is exactly the initialization definition. At each next step *Optimized* growths with the newly optimized state, which was globally constrained at the previous step, as it belongs to *GloballyConstrained.* This state fits the *Optimized* invariant statement since the (1) loop in procedure fulfills the requirement. Regarding *GloballyConstrained* we find that each procedure step adds to this set all the globally constrained states that are not yet optimized (4), and removes from *GloballyConstrained* the (newly) optimized state (5), this is what the invariant asks for.

Using this invariant we prove that the language of the automaton $\mathcal{A}$ is not modified by the procedure. We use an proof by induction on the number of optimized states, the induction step is made at (*inv*) in the procedure, where the invariant holds.

Language preservation proof Let us consider i as the number of already optimized states by the clock algorithm, and prove the optimization of the i^{th} state did not change the language of the automaton.

$i = 0$: the language of the automaton is obviously the same since no changes were made

$i > 0$: let's call s' the state that has just been optimized,
for any transition t that has been modified within this step:

$c \notin Reset(t)$	from (2)
$c \in uprange(Cpred(t))$, c constrains t	from (3)
$c \in uprange(Cpred(k)) \forall k \in outgoing(s')$	from invariant

- For all runs satisfying $upbound(c)$ in $Cpred(t)$ the language is the same.
- For a run that does not satisfy $upbound(c)$ in $Cpred(t)$ it cannot satisfy it in an of $Cpred(k), \forall k \in outgoing(s')$ and will hence be rejected at next step, rejecting this run at current step won't change the words recognized by the automaton. This holds since there are no reset on stable conditions, see 1.3.3

The globally constrained automaton hence accepts the same words as it did before applying the clock algorithm.

Definition

If the globally constrained automaton $\mathcal{A}'$ of an original automaton $\mathcal{A}$ is constrained, then we say that $\mathcal{A}$ was *completely constrainable.* Remember that $\mathcal{A}'$ is constrained iff the stable condition of its unfair states are labelled by a clock predicate which sets an upper bound on (at least) one clock, and this bound is not arbitrary big.

In other words, a *completely constrainable* automaton can be translated into its corresponding *constrained* automaton simply by globally constraining.

Clock algorithm version 2

One can add two optimizations to the above procedure. Firstly, as we already told, we could decrease *upbound*(c) by 1 when adding it to the clock predicate of the newly constrained transition, we denote this by *upbound*$(c)_{-1}$

Secondly one could constrain any unfair states by a clock c such that all its outgoing transitions are constrained by c, except its stable condition. For instance, let us imagine the state s_1 of the timed automaton on page 63 is *not* constrained by y. One could add the constraint $[y < 7]$ to its stable condition without changing the language of the automaton since this predicate has to be satisfied on any of the outgoing transitions of s_1, except its stable condition. It is obvious this modification does not change the language of the automaton since such a state cannot lead to an accepting state without being left, and whenever it is left the timing constraint has to hold. In practice this second optimization is really interesting, since it allows to (explicitly) constrain many more states.

We denote the stable condition of a state $s \in S$ by $SC(s)$ and write the fact a state is accepting through the predicate *fair(s)* that maps any $s \in S$ to true if s is accepting and to false otherwise.

```
$Optimized := \emptyset$
$GloballyConstrained := \left\{ s' \in S \;\middle|\; \begin{array}{l} (fair(s') \wedge constraining\_paths(s',c) \;\vee \\ \left( \neg fair(s') \wedge \bigwedge_{t \in outgoing(s') \setminus SC(s')} c \in uprange(Cpred(t)) \right) \end{array} \right\}$
While $(GloballyConstrained \neq \emptyset)$ do
    Choose $s' \in GloballyConstrained$ with the maximal weight
    $\forall\, t \in incoming(s')$
        if $c \notin Reset(t)$ then
            $Cpred(t) := Cpred(t) \wedge (upbound(c)_{-1})$
            let $s$ be origin(t)
            if $\left( \begin{array}{l} (fair(s) \wedge constraining\_paths(s,c)) \;\vee \\ (\neg fair(s) \wedge c \in uprange(Cpred(t)), \forall t \in outgoing(s) \setminus SC(s)) \end{array} \right)$ then
                $GloballyConstrained := GloballyConstrained \cup \{s\}$
    $GloballyConstrained := GloballyConstrained \setminus \{s'\}$
    $Optimized := Optimized \cup \{s'\}$
```

This procedure can be proved using the same reasoning as the previous one, we won't do it here for the sake of brevity, relying on the reader's intuition to bind the two situations.

4.2.3 New expressiveness theorem

The expressiveness theorem in 4.2 states that any constrained automaton has the same expressiveness whether it is considered as a NFA automaton or as a Büchi automaton.

Using the definition of a completely constrainable automaton in 4.2.2, we propose to extend this equivalence to many more TBAs, and thus to the many more specifications they represent:

New expressiveness theorem *Completely constrainable automata have the same expressiveness whether they are considered as NFA automata or as Büchi automata.*

This new version of the theorem is proved through (1) the definition of a completely constrainable automaton, which can be translated into a constrained automaton and (2) the fact a TBA can be completed with a sink state. These two transformations have been proven to preserve the language of the automaton, respectively in 4.2.2 and 4.1.2.

4.2.4 Efficiency

As we told in 4.1.3, a property expressed by a non-failure accepting timed automaton can be verified efficiently, using state reachability techniques. Therefore any property that is translated into a completely constrainable automaton should be expressed as an NFA automaton rather than a TBA, since the equivalence has been proved in the above section.

In the next section we focus on the description of the set of properties that can be translated into such (completely constrainable) automata, and hence into NFA automata as expressive as the TBA automata corresponding to these specifications. We notice that this class of properties includes most of the properties one could want to verify.

4.3 NFA on the specification level

4.3.1 NFA on TBA level

Regarding timed Büchi automata the expressiveness theorem holds iff the TBA is completely constrainable. This means all its unfair states are globally constrained by at least one clock. In the remainder of this section, we illustrate the same requirements on higher-level specification formalisms.

4.3.2 NFA on Live Sequence Charts level

Every state of the unwound automaton corresponds to exactly one cut of the original LSC, as explained in section 1.4. Such a state is labelled by a clock predicate iff there is any timing annotation or timer added to one of the location of this cut. This location can be either cold or hot.

For a state to be constrained it is sufficient to have *one* timed location in its corresponding cut. For every state of the unwinding automaton to be constrained it would hence be *sufficient* to have one timed location on each cut. Said otherwise, the cut should cross the line of a timer whenever it crosses a hot line (location). By "timed" location we mean either a location annotated with a timing annotation, or a location located between the set and the run out of a timer on the same instance. For instance, the LSC in figure 4.4, whose hot locations are all timed, does fit this requirement and will thus be unwound into a completely constrainable timed automaton. checking.

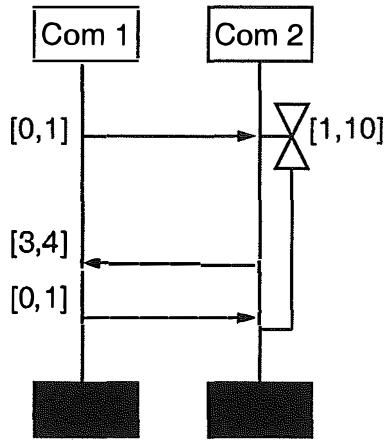


Figure 4.4: LSC suitable for invariance check

Some more considerations have to be taken into account: the cold and maximal locations are accepting, thus if a cut is composed only of cold or maximal locations its corresponding state will be accepting. Such states need not to be constrained. Therefore we should not consider the timing of cold locations into our requirements, only each cut that contains a hot location must contain a timed location.

Finally, the same consideration as the transitivity of clock constraints holds for LSCs. Locations are sometimes depending on each other, because they belong to the same message, condition or simultaneous region, we can hence rely on timing annotations of other instances to deduce information about the occurrence of locations on untimed instances.

Figure 4.5 shows such an LSC, were the hot locations of instance `COM 1` are indirectly covered by the timer of `COM 2`, since the locations on both instance are simultaneous (because the messages are).

More formally one could detail this notion of *time cover* using the simultaneity classes of locations, used in section 1.4.2.

- A location is *directly time covered* either iff (a) it is annotated by a timing annotation, or if (b) it is located between a (re)set and the expiration of a timer related to its instance

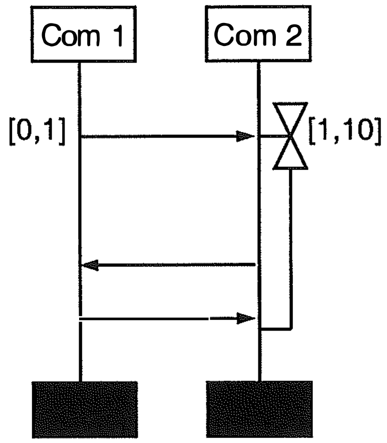


Figure 4.5: More general LSC suitable for invariance check

- A simultaneity class C is *time covered* iff there exists a location $l \in C$ such that l is directly time covered.
- A location is *time covered* iff or it belongs to a time covered simultaneity class.

And therefore we straightforwardly know that any LSC with all hot locations time covered can be translated into a completely constrainable automaton. Allowing an efficient verification of the property it expresses.

4.3.3 NFA on Temporal Logic level

We can use a similar reasoning about temporal logics, but neither LTL nor CTL allow us to easily describe finite-timed properties. [AH94] introduced some annotations, slightly modifying the $\diamond$ (eventually) operator, for it to accept lower and upper bounds. We write that a property p must eventually hold within 2 and 10 steps from here by $\diamond_{[2,10]}p$. The same notation is used for the U (until) operator where the bounds specify the deliverance time, e.g. $\varphi_1 U_{[2,5]} \varphi_2$ stands for $\varphi_1 U \varphi_2$ and φ_2 must hold within 2 to 5 steps from now.

Once these notations are introduced we can express the class of completely constrainable automaton on the temporal logic level simply by saying that all $\diamond$ and U operators of the formula to be checked have to be upper bounded for this formula to be translated into such an automaton.

The explanation is quite intuitive since any fairness constraint in the generated TBA comes from an *until* operator. Constraining it will meet the theorem requirement. The same holds for $\Diamond$ since it is an abbreviation of U.

4.4 Practicability considerations

In this chapter we shown how some particular specifications could be verified efficiently using invariant checking methods, provided that they meet some timing requirements, i.e. if they do not include real (unbounded) fairness.

In the design verification of reactive systems one could almost always have an idea of the reaction times of the system under development. Using real (unbounded) liveness could be avoided most of the time, as any designer who knows his system well should be able to express a reasonable time bound on a sequence of events. Actually unbounded liveness can be found at the first stages of the design, but one can define, i.e. bound, them when the design evolves.

The success of SAT-checkers in the recent time, for instance, which cannot deal with real fairness and approximate infinite behavior on finite observations, shows that in practice one can describe all properties within a finite (bounded) framework. We introduced SAT-based model checking in section 2.4.3.

For this reason we believe that the non-failure criterion is useful since it allows many convenient specifications to be checked using invariant check. This check, it is worth mentioning, improves mainly falsification verifications.

In the next chapter we detail how these improvements were successfully implemented at CvOU. The unwinding automaton we characterized in chapter 1 is optimized and translated into SMI code to be given to the model checker. We therefore review the SMI formalism, and explain our translation.

Chapter 5

Real usage

5.1 Verification environment

We already introduced and described most of the formalisms that are integrated in the verification environment of the Computer Science Department of Carl von Ossietzky Universität, Oldenburg (CvOU). The way they interact is illustrated in figure 5.1 where a new formalism, SMI, is used as intermediate between the already known "high-level" formalisms and the "low-level" model checker finite state machines. SMI is a simple imperative programming language that implements the behavior of (reactive) models by describing the relation between current values and next values of a set of variables. It is explained is section 5.4.

Two situations are depicted in figure 5.1. The current one, in white, differs from the initial one, in light grey, which could not yet handle LSCs.

First, specifications were described using Symbolic Timing Diagrams (STDs), a diagram based language that allows a concise and intuitive formulation of timed properties using collections of (waveform) diagrams [FJ96]. These STDs were unwound into TBAs [Fey96] using a procedure similar to the one for unwinding of LSCs. Actually it is this procedure that inspired [KW01] for handling LSCs. This unwinding procedure also relies on the concept of cut, here called *phases*, which exhibits any possible state of the (STD) property, and are straightforwardly translated into states of the resulting automaton. These TBAs were then transformed into TCTL, a timed variant of CTL, before being finally rewritten into SMI and given to the VIS model checker, after some optimizations.

The figure 5.1 illustrates the current situation of the verification environment, including the support for Live Sequence Charts (LSCs) specifications which has been added recently. Among other improvements, this language allows to distinguish between possible and mandatory behaviors in the requirements (see 1.2). LSCs are translated into optimized TBAs using the unwinding procedure described in section 1.4. The TBAs are then translated into SMI code (5.5), and in BLIF-MV (5.6), the internal VIS formalism.

To generate SMI code we propose, in section 5.2, to optimize the TBAs resulting of the unwinding procedure. We then present the SMI formalism (5.4,

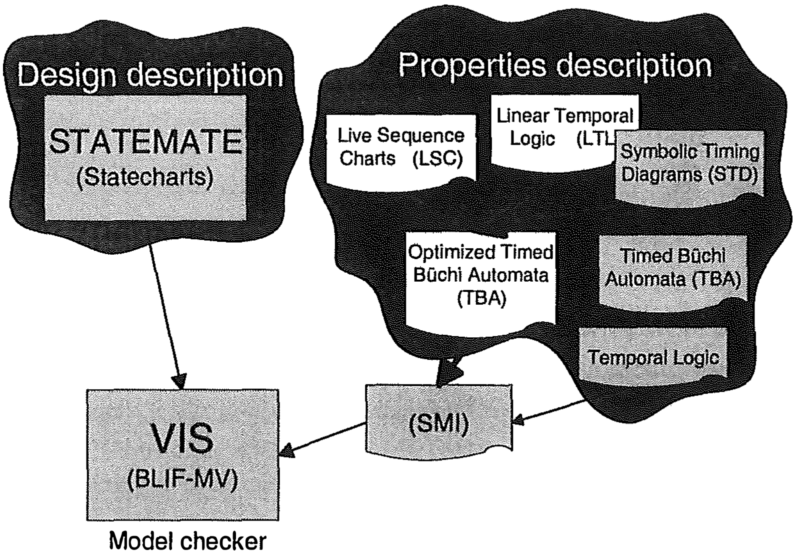

Design description
STATEMATE
(Statecharts)
Properties description
Live Sequence Charts (LSC)
Linear Temporal Logic (LTL)
Symbolic Timing Diagrams (STD)
Optimized Timed Büchi Automata (TBA)
Timed Büchi Automata (TBA)
Temporal Logic
(SMI)
VIS
(BLIF-MV)
Model checker

Figure 5.1: Verification environment at CvOU

and detail the translation process in 5.5. We did not reuse the translation from TBAs to SMI via TCTL that already exists for STDs (in light gray) to handle the LSC way, since the non failure acceptance criterion defined in the previous chapter could not be integrated with this former approach. It is worth to mention that this new translation option can be used for STDs specifications as well, as shown in figure 5.1, enabling the same optimizations for these specifications as for LSCs.

5.2 TBA optimizations

Both the LSC and the STD unwinding procedure, respectively described in [KW01] and [Fey96] produce an automaton that exactly represents the scenario described in the specification, but which is not optimal. We describe the output of these procedures and give some optimizations that could be made.

5.2.1 (non)Determinism

In TBAs which are unwound from LSCs, nondeterminism arises when *isolated conditions* are found. A condition is called isolated if there is no reference point for the valuation of this condition, i.e. no other event (message sending or reception), which we can use to refer to. Since there is no observable reference point, which tells us when to evaluate the condition, the state just before the condition is unwound is simply annotated with **true**, the most nondeterministic label.

No more (explicit) nondeterminism is generated. It is still possible to have "hidden" nondeterminism due to overlapping proposition mappings, e.g. if `msg1` and `msg2` are both mapped to the same design event `e`.

In general, nondeterministic Büchi automata are more powerful (expressive) than deterministic ones, this is why we will not try to determinize the automaton. Such optimizations are nevertheless already available at the SMI level.

5.2.2 Static simplifications

We can encounter some unwinding automaton with inefficient transition (or stable condition) label. We should hence apply the well known logical simplifications, for instance $(a \wedge b) \vee (\neg a \wedge b)$ should be replaced by b. Such propositional reductions are made possible thanks to the concise representation we choose for the transition labels, we should obviously apply them to stable conditions as well.

The automaton generated by the unwinding procedures does never link two states by more than one transition, as cuts are all different. Therefore we do never have to merge many transitions into a single one.

Nevertheless, some transitions could have been labelled by **false** after simplifications, possibly generating non-reachable states which should be removed to reduce the automaton's size.

5.2.3 Fairness

The latter and most important consideration involves fairness. When unwinding a timing diagram or a sequence chart into a TBA, the algorithm determines which cuts must finally be left again, the corresponding states are indeed *not fair*, not accepting.

As explained in 1.4.1 the unwinding algorithm moves a front through the chart in order to determine its *cuts*. Exactly one location of each instance belongs to such a cut. For each of these cuts a state of the TBA is generated. Such a cut may be stable forever iff it is composed only of cold locations or maximal locations. If a cut may not be stable forever it implies a following cut to happen finally. Therefore we mark all the states in the TBA which must be left finally as *unfair*, all the other states are fair states, in the sense that we can stay forever in such a state.

Fairness considerations are a key topic when considering CTL model checking since these model checkers do not support fairness within the given properties. One have to give the fairness constraints beside the specification. They are individually translated into an automaton, running in parallel with the model checking process (see 2.2.3).

In the TBA resulting from unwound LSCs or STDs a lot of fairness is actually bounded, i.e. one expect a fair behavior to occur within a bounded time. This results from the transformation of any timer into a fairness constraint. This bounded fairness is simpler than real fairness. Thus, one could possibly get rid of it.

The fairness optimization is done by application of the clock algorithm (of section 4.2.2)to the automaton. If the globally constrained resulting automaton is constrained we are allowed to verify the property using invariant check, as stated by the expressiveness theorem of 4.2.3. This verification does not take care of the fairness constraints, since they are all bounded fairness constraints.

5.2.4 Goal definition

Regarding the initial situation depicted in 5.1 and the strength and weaknesses of the unwinding procedures, the following aims were defined to improve the former verification environment at CvOU (in light grey in figure 5.1).

- Allow the use of LSC specifications to be used as design properties to be checked.
- Find a solution as deterministic as possible, e.g. using existing SMI tools.
- Get rid of fairness constraints whenever possible.
- Allow real LTL model checking, not ACTL approximation anymore (see 3.4.2).
- Enable iterative activation-mode support
- Extend the witness possibility of the environment, to get a positive witness of the traversal of the automaton, such a purpose is also called *existential verification.*

The creation of the targeted optimizations on the SMI level would have made them available on a wide scale, since SMI is an intermediate language used in many other fields at CvOU. But because SMI is an imperative language there are many different ways to represent the same situation, resulting from implementation choices. Particular optimizations like fairness considerations are hence impossible on this level. This is the major reason why we chose to make them on the TBA level.

After the above described optimizations, including statical simplifications and fairness optimization, we are able to translate the automaton into SMI code.

5.3 SMI translation

In order to perform timed verification using the VIS model checker (see 3.2) both the design and the specification set have to be translated into a formalism interpretable by the model checker, i.e. BLIF-MV (3.2.3).

The model of the system under development is initially specified in the STATEMATE environment using module-, activity- and state-charts (see 3.3.1). Those charts are translated into Finite State Machines (FSM), expressed in the BLIF-MV formalism, directly understandable by the VIS model checker.

The translation of the specification set, which is a main topic of this report, is done in three steps. The initial specifications, in Live Sequence Charts, are translated into optimized timed Büchi automata, this was the issue of chapters 1 and 4. The optimized automata are then first translated into an intermediate language called SMI, a language for the translation of high-level formalisms into FSMs. In the last phase the generated SMI code is translated into a FSM for model-checking, namely BLIF-MV.

In the remainder of this chapter we describe the SMI formalism, in section 5.4, well served by many optimizations, such as determinization. These optimizations are not possible on the BLIF-MV level. We detail in section 5.5.1 the SMI code structure and how our timed automata (of 1.6) optimized by the clock algorithm (of 4.2.2) can be used for invariance checking, since it can be expressed by a NFA automaton (4.1.3). The SMI step gives us the ability to (finally) handle the activation modes (see 1.5), which are introduced into the produced code in section 5.5.2. We finally show in section 5.5.2 how the implementation we choose gives the opportunity of an explicit witness verification[, as suggested by the targets of the verification environment at CvOU in section 5.2.4].

5.4 The SMI formalism

The System Modeling Interface (SMI) can be considered as a general language for describing behaviors. SMI is a simple imperative programming language, containing concepts to model hierarchy, parallelism and nondeterminism.

5.4.1 Syntax

Language constructs

One SMI program, also called *module*, represents the *behavior* of a design. It is composed of one code block containing a single *non terminating loop* to figure the cyclic behavior of a design [Bie01a]. As pictured in figure 5.2[1] the code block of the while loop contains statements. This formalism offers *statements* for assignments, null operations, deterministic branches, nondeterministic branches, "while" loops, breaks and sequential and parallel compositions (not illustrated). No function call nor recursive mechanism are provided.

The branches, or cases, can be either deterministic (DCASE) or nondeterministic (NDCASE). The different branching possibilities are given by *guarded expressions*, which are constrained (guarded) by an expression. The *expression* language contains common boolean and numeric operators, and selection on arrays and records. A deterministic branch allows no overlapping between the guards of its guarded expression and activates the expression whose guard is evaluated to true. A nondeterministic one chooses between all true valued expressions.

Typology and variable definition

The supported *data types* include bit, integer (bounded or unbounded), real (bounded or unbounded), string, and enumeration.

These basic types can be used within *aggregated types* like records, unions, arrays or queues. Besides these, a special type, called *reference*, can be used to declare aliases to existing types.

Any variable in SMI code is characterized by its name, type, group, mode, initial value, and some additional information we do not detail here, as an internal name and the method it (possibly) uses to access memory [Bie01b]. The

[1] This picture is inspired from [Bie01a]

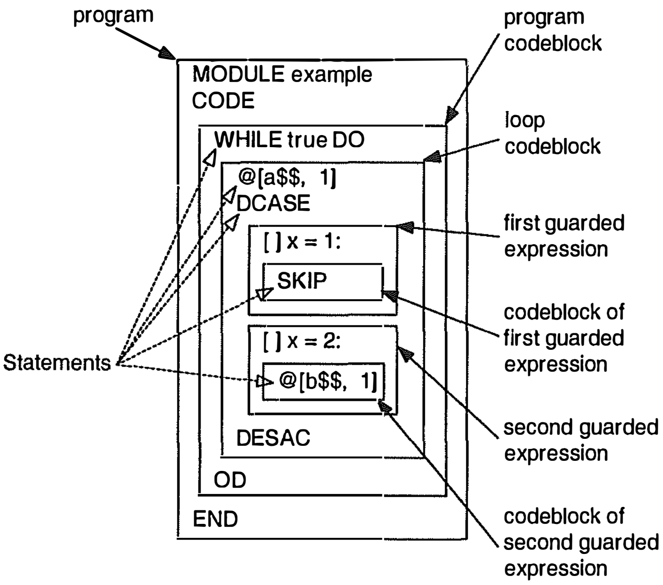


Figure 5.2: The structure of a SMI program

group of a variable depicts the role of the variable within the FSM, these roles include data items, events, conditions and timers. The *mode* ranges between input, output, local, observer and constant, among others. A local variable is particular to the concerned module, while an observer variable could be considered as a token of a defined behavior. The other modes are self-explaining. These additional variable information are not really constraining, but refine the way produced counter-examples are interpreted.

5.4.2 Semantics

Step semantic

SMI implements the behavior of (reactive) models and (timed) properties by describing the relation between current values and next values of a set of variables, using a *synchronous step-semantics* idea.

In SMI all control information, variables and events of the design are encoded into variables. SMI provides two versions of all the variables present within the code, the primed and the unprimed. The *primed* variables are used to express the values of these variables in the next step, while the *unprimed* ones carry the value of the variable at last step. There is an implicit copy-action when the complete SMI-Program has been executed, where the unprimed versions get the current values of the primed one in order to get a "stepping system". Assignments are allowed to primed variables only.

The cyclic behavior of a design is expressed as a non-terminating loop in SMI code, following Misra and Chandy's Unity model [CM88]. One execution of this loop corresponds to exactly one step of the design, whether considering step semantics or the super-step semantic defined on STATEMATE designs (see section

3.3.2). Both of these semantics are supported by the verification environment we use here.

Conventions

By convention, actually to allow an easy portability with other tools of the environment, the name of variables representing clocks must be of the form znumber, e.g. `z00` or `z312`. Similarly we rename all the states so that their names begin with `X`.

5.4.3 Propositional architecture

One purpose of the SMI environment is to allow a great modularity of each of its components. Therefore the information of a single specification (or model) is split into many parts, each corresponding to a particular information type. We hence distinguish

- The type definitions and variable declarations, which are put together into a *symbol table.*
- The SMI program, the real finite state machine, as described above, any variable used within the code should be defined in the symbol table.
- The mapping of each variable of the program (declared within the symbol table) to the corresponding item (event, proposition, trigger,...) of the real STATEMATE design. This mapping is written into a *proposition table.*

This high modularity allows specific optimizations and enhance the reusability of partial information. One could for instance reuse the same proposition mapping for many specifications, or, at contra, reuse the same specification with different variable definitions and/or propositional mapping.

In the remainder of this chapter we don't distinguish anymore between the SMI code, the symbol table or the propositional table, as it is clear form the context which part we're considering. For instance variable declaration always belongs to symbol table, including all the mentioned additional information, whereas guarded expressions and code block always refer to the SMI program.

5.4.4 Available optimizations

The SMI format is well served by manipulations and optimization tools, the tools within parentheses provide some valuable optimizations on SMI code including

- making it more deterministic (`smidet`)
- computing the cone of influence (`smicoi`). Such abstractions achieve a further reduction of FMS complexity, permitting the checking of still larger designs. The cone of influence abstraction is described in section 2.4.2.

Besides these exact optimizations there are also approximations that can be made. Over-approximations include abstraction, freeing variables, fixpoint approximations. Under-approximations include freezing inputs to a constant value and removing nondeterminism, among other, and are used mainly for witness-based model checking.

5.5 Translation of TBAs into SMI

The translation process performs several tasks. It maps used data-types onto the types of SMI. The state configurations of the chart are encoded into SMI variables. Therefore the TBA2SMI tool defines variables to encode data, boolean variables, events, and the control information of the automaton. With these variables we keep some additional information for traceability information (data-type, mode and group), which are used by the model-checker to clarify a counterexample for a given property.

Time consideration

To cope with timing aspects of a specification the translation process introduces clock variables, all running synchronously. Because we require all time expressions to evaluate to a constant at compilation time finite domains for the clocks can be determined.

Details on generated SMI

Three specification activation modes are supported: initial, invariant and iterative. None of these modes influences the TBA optimizations applied in section 4.2.2, they simply express the range of the specification, depending on their meaning we explained in section 1.5. As the SMI code should express the whole specification, including its activation condition, we detail their handling in section 5.5.2.

We refine the SMI output into two main parts: an *activation part*, that handles the activation mode, followed by the *core automaton*, that represents the TBA's transitions.

5.5.1 Core automaton

Generation mode

One single run on the automaton has one single active state at the time. We can represent this behavior by mapping one boolean variable to each state, one of each can be active at a time, and the only active state (variable) can possibly change at each step. This way of implementing an FSM is called *one-hot* encoding, since one state is active (hot) at a time.

On the other hand one could model this behavior of "one state active at a time" using a single integer variable, whose value could change accordingly to the active state. This *logarithmic* encoding sets an "active state" counter to the value of the current state, which is simply an integer bounded by the number of states in the automaton. This latter approach is more efficient than the one-hot encoding since fewer variables are allocated, nevertheless it provides a less readable result. Therefore we will illustrate our examples using only the one-hot code generation mode, even if both are available in the TBA2SMI tool.

Before giving the complete translation of a TBA let us first review how its constituting elements should be handled, according to the possibilities given by SMI (in 5.4) on one hand and the semantics of TBAs (in 1.6) on the other. All these elements are illustrated in the SMI code of figure 5.4, on page 85.

State space

Depending on the chosen code generation mode we declare either many *boolean* variables, or one *bounded integer* one to represent the state space. These variables are actually control variables of the FSM, which should only be modified within the module. They are thus declared within the symbol table (see 5.4.3) as *local data items* since their values are allowed to change in time, referring to the typology we gave in 5.4.1.

The state variables in figure 5.4 are `X1` and `X2`.

Atomic propositions

Any atomic proposition of the automaton to be translated is declared as a *input boolean event*. This means the value of an atomic proposition can be either true or false, and is determined at any moment by a factor external to the specification, i.e. the model. This behavior is exactly what we are expecting from an atomic proposition.

The atomic propositions variables in figure 5.4 are p1 and p2.

Clocks

Clocks are internal to the FSM, which should be allowed to modify their values, therefore we consider *local bounded integer data item* variables to represent them. The bound on their domain can be computed statically relying on the hypothesis of (non arbitrary big) upper-bound that stands for any clock of the TBA.

In both modes all the clocks are increased by one before each step (at least while they remain within their domain), at the very beginning of the body while loop, and possibly reset by a transition after this one has been taken. The only clock variables in our sample code is `z00`.

Fairness

A run is fair (accepting) in Büchi's sense if it infinitely often passes through an accepting state (see 1.3.1). We represent this by creating an *observer* variable which is true iff the actual state is accepting, checking for acceptance is then reduced to check whether this variable is infinitely often true. Because we defined it as an observer it can be accessed from outside the FSM, i.e. by the model checker, to verify its value.

We call this observer the *fairness observer*, since its true evaluation states the current state is fair, as shown in figure 5.4 with variable `fairness_observer`.

Sink state

We showed in section 4.1.2 that the addition of a non-accepting sink state, reached at any step is any other transition is taken, does not change the expressiveness of the automaton.

We introduce this sink state explicitly at the SMI level, rather than before (at the TBA level), because we can use an efficient implementation artifact, saying the sink state is always active (or reached), except if any other one is. The corresponding variable, `failure`, is initialized to true, and set to false if any transition is taken.

Transition relation

We call the encoding of the transition relation the *core automaton*, or core TBA, with respect to the activation part, which we detail below. The core automaton depends highly on the SMI *generation mode* chosen.

The figure 5.4 shows the SMI code of the sample TBA given in figure 5.3 in one-hot encoding mode.

Each guarded expression represents exactly one transition of the automaton. For a transition t of the automaton we can find exactly one guarded expression whose guard is the conjunction of the origin state of t, stating it is the current state, and the transition label. The codeblock of a transition updates the active state variables of both the left and the reached states, and asserts we are not in the sink state, by setting the (primed) value of `failure$$` to false.

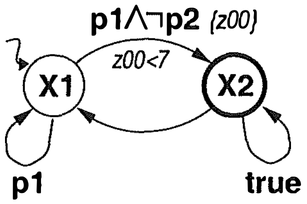


Figure 5.3: A simple TBA.

5.5.2 Activation part

The three different activation modes we consider were defined in section 1.5. These activation modes describe the range of the property expressed by the core TBA, e.g. whether it should hold immediately (initial mode), forever (invariant mode) or forever with no overlapping (iterative mode).

One can consider that if the activation condition is not met (and thus the TBA never activated), one can accept the run as fulfilling the specification. This is implemented by setting the fairness observer to true by default, as show in figures 5.5, 5.6 and 5.7 for each of the activation modes. One can also want to force the property to be verified, this behavior is called *healthiness* in [DH98]. One could for instance require healthiness of a specification on the assumption

```
MODULE sample_lsc
CODE
WHILE true do
 DCASE
  [] z00 < 7 :
     @[ z00$$, z00 + 1 ]
 DESAC
 @[ failure$$, true ]
 NDCASE
  [] (X1$$) and (p1) :
    @[ failure$$, false ]
    @[ X1$$, false ]
    @[ X1$$, true ]
  [] (X1$$) and (p1 and not p2 and (z00$$ < 7)) :
    @[ failure$$, false ]
    @[ X1$$, false ]
    @[ X2$$, true ]
    @[ z00$$, 0 ]
  [] (X2$$) and (true ) :
    @[ failure$$, false ]
    @[ X2$$, false ]
    @[ X2$$, true ]
 NDESAC
 @[ fairness_observer$$, false ]
 DCASE
  [] X2 :
    @[ fairness_observer$$, true ]
 DESAC
OD
```

Figure 5.4: SMI code: the core TBA

side, to prevent the model checker from never activating the specification, satisfying it on the easiest way. Therefore the TBA2SMI tool provides the ability to rather initialize the fairness observer to false.

Whatever the code generation mode, one-hot or logarithmic, we introduce two new control variables to handle the activation modes. One is called `tba_started`, it allows us to distinguish between an initialization pass through the body loop or a step on the core automaton. Since this variable is internal to the FSM and has to be updated we declare it in the symbol table as a *local data item boolean* variable, using the topology of section 5.4.1. We put the core tba within a guarded expression guarded by the true evaluation of `tba_started`, while the activation part needs it to be false to be traversed. The second variable is the activation condition variable, declared as a *boolean input event*, just like any atomic proposition, it is called `ActivationCondition` in our example code.

The remainder of this section explains how the different activation modes are integrated within the SMI code, by adding a code fragment at the very beginning of the body while loop, before the clock's increment `DCASE` of figure 5.4.

Initial mode

A run on an initial TBA has to fulfill immediately, i.e at the first step of the run, the *activation condition* and be fair to be accepted. It is also accepted if the activation condition does not hold at first step while the *activation exception* does. In this case the core TBA isn't even activated, and the run immediately succeeds (see 1.5.1). If the activation condition is not satisfied at the first step, and neither the activation exception is, then the run is rejected.

The "exception" mechanism leads us to the definition of a corresponding new acceptance token, which states explicitly that the exception has been used. In practice we could also use the same observer than for the Büchi criteria (i.e. `fairness_observer`). We introduce a trigger (a *local data item boolean*) to ensure the evaluation of both activation condition and exception occurs only at first step, it is called `first_step` in figure 5.5, and is initialized to true. Obviously, once the core automaton activated, the acceptance of the run relies on infinitely many true evaluations of the fairness observer, as stated in section 5.5.1.

Invariant mode

An invariant activation mode means the run has to satisfy the automaton *any time* its activation condition holds.

This invariant mode can be handled in different ways, from which the worse is perhaps building its product automaton. The TBA2SMI tool produces a *non-deterministic activation* of the core tba[2] as shown in figure 5.6

[2] This way of handling multiple activations is adapted to the VIS model checker, which is used to check the model later on.

```
DCASE
 [] not ( tba_started ) and ( first_step ):
   @[ fairness_observer$$, true ]
   @[ first_step$$, false ]
   DCASE
     []  ActivationCondition :
       @[ tba_started$$, true ]
       @[ X1$$, true ]
       @[ z00$$, 0 ]
     [] not ( ActivationCondition ) :
       DCASE
         [] ActivationException :
           @[ initial_accept$$, true ]
         [] not ( ActivationException ) :
           @[ failure$$, true ]
           @[ fairness_observer$$, false ]
       DESAC
   DESAC
DESAC
```

Figure 5.5: Activation in initial mode

```
DCASE
 [] not ( tba_started ) :
   @[ fairness_observer$$, true ]
   NDCASE
    [] ActivationCondition :
       SKIP
    [] ActivationCondition :
       @[ tba_started$$, true ]
       @[ X1$$, true ]
       @[ z00$$, 0 ]
   NDESAC
DESAC
```

Figure 5.6: Activation in invariant mode

Iterative mode

The iterative activation mode provides multiple activation of the TBA in the same run, but only one at the time. It uses a *lock mechanism*, described in section 1.5.3, to allow the TBA to be reactivated when its previous activation is completed, even when considering infinite runs. Let us just remember that if a run reaches a so-called *finitely accepting state* it can be considered as complete and hence allow the TBA to be reactivated.

The first activation of the TBA occurs when the activation condition is met for the first time, if it is never the case the specification is accepted. Regarding the possible re-activations, the TBA2SMI tool introduces back-leading transitions from any of these true fair states (X2 in the sample TBA of figure 5.3) to the initial node, and modifies the transitions predicates accordingly, as explained in 1.5.3. The first activation and the modified transitions can be found in figure 5.7.

```
DCASE
  [] not ( tba_started ) :
  @[ fairness_observer$$, true ]
    DCASE
      [] not ( ActivationCondition ):
        SKIP
      [] ActivationCondition :
        @[ tba_started$$, true ]
        @[ X1$$, true ]
        @[ z00$$, 0 ]
    DESAC
DESAC
```

```
[] (X2$$) and ( ActivationCondition ) :
  @[ failure$$, false ]
  @[ X2$$, false ]
  @[ X1$$, true ]
  @[ z00$$, 0 ]
[] (X2$$) and (not ( ActivationCondition )) :
  @[ failure$$, false ]
  @[ X2$$, false ]
  @[ X2$$, true ]
```

Figure 5.7: Activation in iterative mode:
first activation and *reactivation* from any true fair state

Non-failure acceptance criteria

The run of the automaton is fair if it evaluates infinitely often the *fairness observer* to true, recalling the Büchi acceptance criterion.

The non-failure state acceptance (see 4.1.2) eventually holds. This criteria accepts the same runs than Büchi's does iff all unfair states of the automaton are constrained by at least one clock. A property represented by a NFA automaton can be computed efficiently using invariant check methods (see 4.1.3).

If the hypotheses of this criterion hold, i.e. if the automaton to be translated is constrained, we define a non-failure acceptance *observer* in the generated SMI code, which becomes false whenever the sink state (`failure`) becomes active. The once false value of this `non_failure_observer` is sufficient to reject a run.

The two fairness mechanisms are illustrated on the sample TBA in figure 5.8, where `fairness_observer` is true whenever the active state is accepting, while the `non_failure_observer` becomes false if we reach the failure state.

We are allowed to illustrate both criteria since the TBA of figure 5.3 meets the requirements of the non failure acceptance condition.

```
@[ fairness_observer$$, false ]
DCASE
  [] X2 :
    @[ fairness_observer$$, true ]
DESAC
DCASE
  [] failure$$ = true :
    @[ X1$$, false ]
    @[ X2$$, false ]
    @[ non_failure_acceptance$$, false ]
DESAC
```

Figure 5.8: Two observers, for two acceptance criteria

Witness mode

To provide an existential verification of any property we rely on the finitely accepting states concept (in 4.1.2).

Since any run that reaches such a state in the TBA is (definitely) accepting we can, for instance, introduce a new observer that evaluates to true whenever any finitely accepting state is reached, let's call it `witness_observer`. An existential verification would then simply consist of a falsification of the formula $\Box\ (\neg \mathit{witness\_observer})$ that would provide a counter-example of a complete traversal of the automaton.

5.5.3 Correctness

Since the translation of TBA into the corresponding SMI code is quite intuitive, thanks to both formalisms, and highly detailed in the above sections, we won't prove more formally the correctness of our implementation choices.

5.6 Final steps before model checking

We now sketch the translation of SMI into BLIF-MV, the internal formalism of VIS (see 3.2.3) . This translation entails BDD manipulations, which we do not want to detail in this report, hence we only sketch the guide lines of this final step.

The translation generates BDDs, a characteristic function (see 2.4.3) is computed for every bit of the state space, i.e. the variables defined in the SMI code. No additional variable is needed to represent the locations since we use a step semantics (one step of the FSM is one complete run through the nonterminating outermost loop of the SMI program).

Some additional input variables have to be added to cope with *nondeterminism.* These "choice" inputs are actually chosen between the possible runs through the SMI program, and hence resolve the nondeterminism.

The *loops* can be handled either by computing a fixed point for the loop (very slow) or by unrolling the loop. The latest method does not fit endless loops, but is faster whenever usable. We cannot state in advance wether the body while loop will belong to one or the other of these two categories, since it depends on the code it executes.

Because after the translation of a design and properties into SMI, all necessary clocks are represented by a finite number of bounded variables (see 5.5), one can generate untimed FSMs from the SMI code. The constructed FSM is such that one step of the FSM corresponds to one execution of the complete while-loop of the SMI code. Thus, in step semantics the FSM timers are increased by one in each state of the FSM. In super-step semantics timers are increased only in certain states, while they remain unchanged in all other states.

5.7 Conclusion

In this chapter we presented the SMI formalism and a possible way of translating TBAs into programs of this language. Many other possibilities are allowed since SMI is a simple imperative program, allowing the same behavior to be represented on many different ways.

In the next, and last chapter, we highlight the improvements brought to the CvOU verification environment through the introduction of this new translation.

Chapter 6

Results

The prototypical development and integration of the translation tool described in chapter 5 has been conducted from September 2001 until December 2001 at the Carl von Ossietzky Universität, Oldenburg, Germany (CvOU).

We detail in this last chapter some improvements this new verification "option" brought to our Embedded Systems department verification environment. One should compare them with the goals that were identified to enhance our environment, on page 78.

6.1 Specification support

6.1.1 LSC

Live Sequence Charts specifications can be used as well as STDs to specify properties one could want to verify. We detailed in this report how such a specification is first unwound into a timed automaton (chapter 1), translated into SMI code (chapter 5), represented as a BLIF-MV finite state machine (5.6) and given to the VIS model checker to be tested on a design under development modeled with STATEMATE (chapter 3).

This translations chain allows *any* LSC specification to be verified on a design under development using symbolic CTL model checking with fairness constraints, based on BDDs techniques (chapter 2).

Nevertheless some approximations are made (3.4.2) which sometimes prevent us from being able to endorse a design, since some properties can be invalidated because of the approximations rather than because of the design errors.

6.1.2 LTL

Linear temporal logics can be checked the same way as LSCs, as long as the automaton corresponding to the formula fits the requirements of our TBA format.

We cannot hence offer support for the whole LTL formalism, but detailed in section 3.4.1 the subset of LTL we can verify. Actually we don't allow any nested $\Box$ (Globally) operator, to guarantee the acyclicity of the corresponding automaton.

The translation of LTL into automaton is an active field of research in formal methods, as the state explosion problem described in 2.3.2 is still a challenge. [SB00], among others, gave one way to generate a reasonably small Büchi automaton from an LTL formula. This translation is not implemented yet in our verification environment, but will be soon since it is part of the next VIS release (VIS 2.0).

The choice between a linear or a temporal framework, sketched in 2.3.1, remains a current issue. Our choices, i.e. a linear specification framework on a branching-time verification core meets our present needs. Mixed solutions, as the one we describe here seem to be one path through this topic.

6.2 More efficient verification

Many specifications are accurate enough to be verified on a more efficient way than fair CTL model checking.

We integrated the non-failure acceptance criterion (defined in chapter 4) in the new verification chain. Properties represented by a NFA automata can be verified efficiently using invariant check techniques, i.e. on-the-fly state reachability analysis.

Actually not all properties can be verified using this technique, since heavy requirements are needed for a property to be represented as a NFA automaton. But we suggested in 4.3 that many "real" specifications could meet them.

Above this, the invariant check gives us a possibility, again for the same specifications only, to verify them even if the approximated fair ACTL model checking fails. Hence, we are sometimes able to assert or reject a property which we could not verify before the NFA criterion integration.

6.3 Iterative activation mode

We gave in 1.5.3 the definition of the iterative activation mode, and showed in 5.5.2 how we implemented it within the design verification process.

This activation mode is new within our verification environment. We believe it to be useful since it provides a *deterministic* activation of the property check, and permit many (successive) activations in addition. The fact it is deterministic is quite interesting since nondeterminism is hard to handle in the model checking process.

6.4 Witness verification

We allow witness verification tests to be conducted. This generates a positive witness of the traversal of the automaton. We explained in section 5.5.2 how we implemented it. Such a verification is reduced to an invariant check, hence as efficient as the on-the-fly reachability states computation we already talked about.

Chapter 7

Conclusion

One of our goals was to introduce the reader to the field of *formal verifications*, as well theoretical as practical, since the first would make no sense without the latter, whereas the second could not progress that much with no theoretic fundamentals.

In particular we defended the use of timed *Live Sequence Charts* as formal specification language. We believe the graphical appeal of this formalism coupled with its intuitive syntax and interesting features make it user-friendly and powerful. Temporal behaviors of reactive systems are especially the kind of scenario this formalism is suitable to describe.

We then detailed how these specifications could be verified on modeled designs, using either fair CTL model checking or automata-theoretic related techniques. We first sketched the intuitions beyond these approaches, giving some related current issues and optimizations as well. We then showed how such a verification can be conducted in practice, using the VIS model checker and STATEMATE designs.

On our way towards efficient validation we defined a property class that can be verified efficiently, using invariant check, i.e. an efficient state reachability analysis. To formally define this class of specifications we introduced a new acceptance criterion on timed automata. We proved that, under certain circumstances, the expressiveness of a timed automaton is the same whether considered with the Büchi acceptance condition or with the *non-failure acceptance* one. We admitted that the requirements for this equivalence to hold are quite constraining, but we also showed that most properties one could want to verify in real design development usage should meet them, hence our interest in this improvement.

Moreover, we gave in this master thesis both theoretical and practical justifications to these considerations. Giving a formal hierarchy of the clock *constraints* on the theoretical hand and an intuitive *algorithm* to implement the new acceptance criterion on the practical hand. We proved the algorithm preserves the language of any automaton it optimizes.

The field of formal verification is currently focussed on some major topics, one of which is the choice between a linear or a branching time framework. Many initiatives are taken to compose with the advantages of both approaches, and our works situates itself on the same path. We motivated our choices for a linear time specification framework coupled with a branching time model checker.

We are aware our notion of time is far more simple than some real-time considerations [AD94] which actually impassion many formal verification research groups. Nevertheless we showed interesting things could be done using a (simplified) discrete-time framework, which often suffices to model the target design.

One could pretend that some of our requirements are quite constraining, as, for instance, the obligation for our TBAs to be acyclic. We explained why we use them and our reason are, almost always, empirically justified. Nevertheless from a theoretical point of view many new problems could arise if we remove some. We pretend that, even if a problem seems interesting on the theoretical point of view, there is only few interest in resolving it if we know in advance our solution won't be used by anyone, i.e. industrially speaking.

Some interesting researches could be conducted to further investigate the possibilities we described here. For instance one could improve the iterative activation mode, as sketched in 1.5.3. Another further-work topic would be the looking for better explicit clock constraints definition, which would constrain more labels without modifying the language of the automaton.

Bibliography

[AD94] Rajeev Alur and David Dill. A theory of timed automata. *Theoretical Computer Science*, 1994.

[AE01] Amyot and Eberlein. Evaluation of scenario notations for telecommunication systems development. In *Proceedings of 9th International Conference on Telecommunication Systems (9ICTS)*, Dallas, USA, mar 2001. ICTS.

[AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, 1994.

[Alu97] Rajeev Alur. Timed automata. In *NATO*, 1997.

[BCC$^+$99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[BCM$^+$90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[BF93] Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. In *IEEE Transactions on Software Engineering*, volume 19, pages 3–12. IEEE, Jan 1993.

[BH] Yves Bontemps and Patrick Heymans. Turning high-level live sequence charts into automata. In *Proc. of the 24th International Conference on Software Engineering (ICSE 2002): Scenarios and State Machines: algorithms, models and tools.*

[BHSV$^+$96] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, 1996. Springer Verlag.

[Bie01a] Tom Bienmüller. Simlib referenz, v0.7. Technical report, Carl von Ossietzky Universität, Oldenburg, Germany, Sep 2001.

[Bie01b] Tom Bienmüller. Symlib referenz, v0.7. Technical report, Carl von Ossietzky Universität, Oldenburg, Germany, Oct 2001.

[BP94] François Bodart and Yves Pigneur. *Conception assistée des systèmes d'information. Méthodes informatiques et Pratiques des systèmes.* Masson, Paris, 2 edition, 1994.

[Bra94] R.K. Brayton. HSIS: A BDD based system for formal verification. In *Proc. of Design Automation Conference*, 1994.

[Bro86] Brooks, Frederick P., Jr. *No Silver Bullet: Essence and Accidents of Software Engineering.* Number ISBN No. 0444-7077-3. Elsevia Science Publishers B.V., North-holland, 1986.

[CGP99] Clarke, Jr., Edmund M., Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, Massachusetts Institute of Technology, Cambridge, Ma 02142, 1999.

[Che91] Szu-Tsung Cheng. Compiling verilog into automata. Master's thesis, Computer Science Division, Departement of Electrical Engeneering and Computer Sciences, University of California, Berkeley, CA 91720, 1991.

[CM88] K.M. Chandy and J. Misra. Parallel program design: A foundation. *Addison-Wesley*, 1988.

[DGV99] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear time temporal logic. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, number 1633 in LNCS, pages 249–260, Berlin, 1999. Springer-Verlag.

[DH98] Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Diagrams, 1998.

[Eme90] E. Allen Emerson. *Temporal and Modal Logic*, volume B, chapter 16, pages 997–1072. MIT Press and Elsevier Science Publishers, Cambridge, Massachusetts, 1990. ISBN 0-262-72015-9.

[EOK94] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 415–427, Standford, California, USA, 1994. Springer-Verlag.

[Fey96] Konrad Feyerabend. Real time Symbolic Timing Diagrams. Technical report, Carl von Ossietzky Universität Oldenburg, 1996.

[FJ96] Konrad Feyerabend and Bernhard Josko. A visual formalism for real-time requirement specifications. Technical report, Carl von Ossietzky Universität Oldenburg, 1996.

[Gil97] Tom Gilb. Towards the engineering of requirements. *Requirements Engineering*, (2):165–169, 1997.

[GO76] Gries and Owicki. An axiomatic proof technique for parallel programs. *Acta Informatica*, pages 319–340, 1976.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[HN96] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions of Software Engineering Methods*, 5(4):1–36, Oct 1996.

[Hoa69] Hoare. An axiomatic basis for computer programming. *ACM*, 12(10):576–580, 583, 1969.

[Hol97] C. Michael Holloway. Why engineers should consider formal methods. In *16th AIAA/IEEE Digital Avionics Systems Conference*, volume 1, October 1997.

[HP98] David Harel and M. Politi. Modeling Reactive Systems With Statecharts : The Statemate Approach, 1998.

[HU77] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, USA, 1977.

[Hyd95] Daniel C. Hyde. *CSCI 320 Computer Architecture Handbook on Verilog HDL.* Bucknell University Lewisburg, PA 17837, 1995.

[I-L00a] I-Logix, Inc. Certifier user guide: Pattern library. part from the STATEMATE Magnum help, I-Logix Inc., 3 Riverside Drive, Andover, MA 01810, 2000.

[I-L00b] I-Logix, Inc. Statemate MAGNUM brochure. I-Logix Inc., 3 Riverside Drive, Andover, MA 01810, 2000.

[IT96] ITU-T. Itu-t recommandation Z.120 : Message Sequence Chart (MSC), 1996.

[JRB99] Ivar Jacobson, James Rumbaugh, and Grady Booch. *The Unified Modeling Language reference manual.* Object Technology Series. Addison-Wesley, 1999.

[Kuk96] Yuji Kukimoto. *BLIF-MV.* The VIS Group, University of California, Berkeley, 1996.

[KV98] Orna Kupferman and Moshe Y. Vardi. Relating linear and branching model checking. Technical Report TR98-301, 1998. To Appear In Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98), Shelter Island, NY, USA, June 1998. Chapman & Hall, 1998.

[KW01] Jochen Klose and Hartmut Wittke. An automata based interpretation oflive sequence charts. *TACAS*, 2001. University of Oldenburg, OFFIS.

[Mur90] Dick Murray. Tube train leaves... without its driver. *London Evening Standard*, Apr, 12 1990.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proc. 18th Ann. IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.

[RS99] K. Ravi and F. Somenzi. Efficient fixpoint computation for invariant checking. In *iccd*, pages 467–474, Austin, TX, oct 1999.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient Büchi automata from LTL formulae. In Springer Verlag, editor, *Twelfth Conference on Computer Aided Verification*, number LNCS 1633, pages 247–263, 2000.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal ACM*, (32):733–749, 1985.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191, Amsterdam, 1990. MIT Press and Elsevier Science Publishers.

[TM91] D.E. Thomas and P.R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Nowell, Massachusetts, 1991.

[Var98] Moshe Y. Vardi. Sometimes and not never re-revisited: On branching versus linear time. In *International Conference on Concurrency Theory*, pages 1–17, 1998.

[Var01] Moshe Y. Vardi. Branching vs. linear time: Final showdown, 2001.

[Wol98] Pierre Wolper. The algorithmic verification of reactive systems, 1998. Universitė de Liège, Francqui Chair Lectures Given at the FUNDP.

[WV94] Pierre Wolper and Moshe Y. Vardi. Reasoning about infinite computations paths. *Information and Computation*, 115(1):1–37, 15 1994.

Index

Appendix A

LSC unwinding

This appendix illustrates the complete translation chain we defined in this master thesis. We refer here to a train crossing control system. This system is composed of lights, barriers, and sensors to detect the arrival and the departure of a train.

The specification given by the LSC of figure A.1 describes the first part of a communication, which should take place when a level crossing is secured, so that an approaching train can safely pass it. The LSC is activated when the train announces its arrival, i.e. by sending the message `activate`, which is the activation condition.

The crossing controller (instance `XingCtrl`) reacts by sending an acknowledgement (message `ack`) to indicate that it has received the request and simultaneously it initiates the securing procedure by first ordering the sub-controller for the traffic lights (instance `Lights`) to turn on the lights (message `turn on`).

The lights controller does just that and first switches on the yellow light (message `switch2yellow`), which has to be on for some seconds, before the light changes to red (message `switch2red`), indicating that the car traffic must stop. We model this through the sending of these messages to the environment (instance `XingEnv`). If all this has happened, the lights controller informs the crossing controller that the lights are now on (i.e. red, with the message `lights on`). This communication sequence can of course only happen this way, if the lights are not broken or malfunctioning. This is ensured by the local invariant `no_red_err`. We didn't introduce local invariants in the LSC formalism described in section 1.2.2, since it can also be seen as a triggering message sent by the environment (or any other component), stating the assertion is met.

After completing the behavior shown in this LSC the crossing controller starts a similar interaction with the controller for the barrier in order to lower the barrier, which completes the securing procedure.

The property expressed by the LSC of figure A.1 hence states that every time a train arrives the lights are turned on, since the LSC mode is `INVARIANT`.

We declared in 4.3.2 most interesting properties could be timed "enough" to allow an efficient verification of the property they express. This "enough" means that all hot locations have to be time covered. We hence refine this specification, stating the light should stay yellow for 2 or 3 steps, and the whole

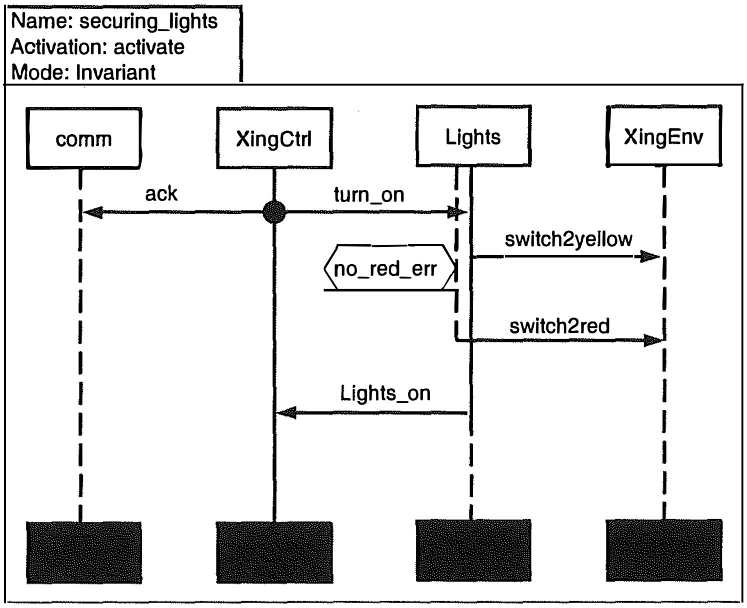


Figure A.1: A LSC property of the crossing controller

securing process should take at most 10 steps to be completed. The resulting LSC is shown in figure A.2.

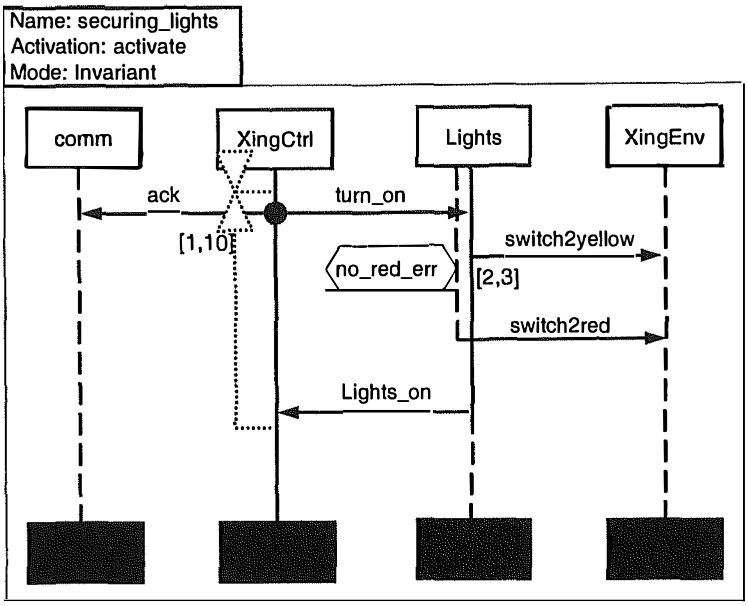


Figure A.2: Adding timing annotations to an LSC

This scenario is unwound into the corresponding TBA (figure A.3) by a strict application of the unwinding procedure of [KW01] which has been described in section 1.4.

The initial state of the unwound TBA is X1. The only acceptant one is the double circled X5. The three remaining X2, X3 and X4states are neither initial, nor accepting.

As explained in section 1.2.2 the cold locations, depicted in dotted portions of the instances lines, represent a state of the instance were the system is allowed to stay forever. We hence see on picture A.3 the only accepting states corresponds to the cut which contains all maximal locations. Since no other cut is composed only with cold and maximal locations.

Recalling the notions of clock *constraining* a state and a transition we can say the clock z0 *constrains globally* all unfair states, or, said otherwise, this TBA is completely constrainable.

The application of the translation procedure described in 5.5 to the specification represented by this TBA generates the SMI code given in figure A.4. One can easily remark the way the invariant activation mode is handled.

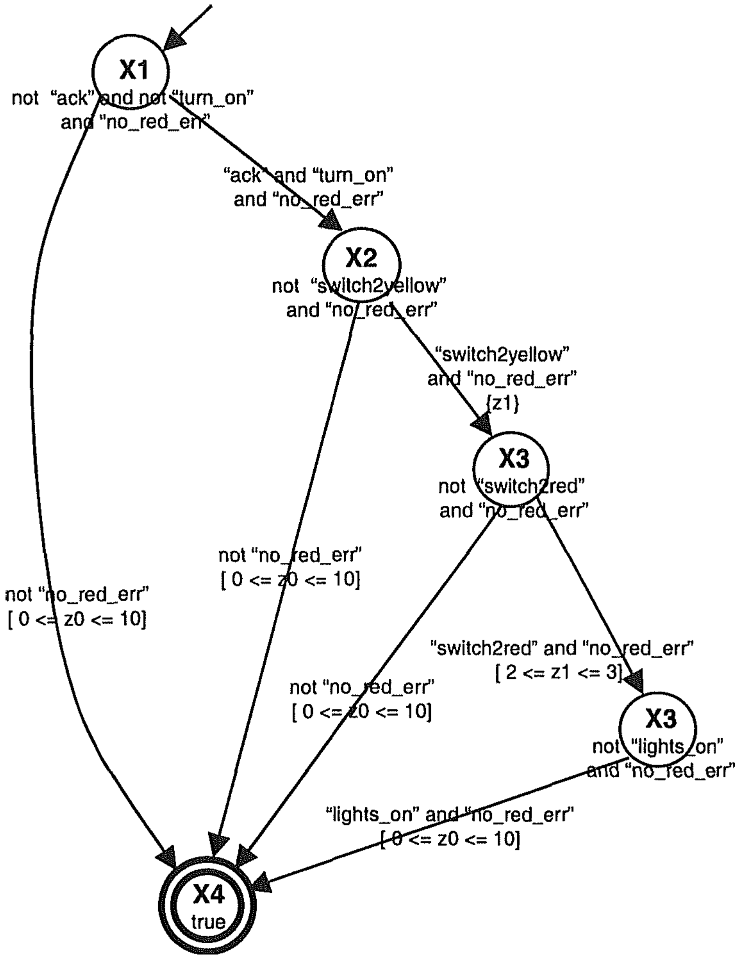


Figure A.3: The unwound TBA

e

```
MO DULE lsc_securing_lights

CO DE
 WHILE true DO
   DCASE
     [ ] not ( tba_started ) :
       @[ fairness_observer$$, true ]
       NDCASE
         [ ] activate :
           SK IP
         [ ] activate :
           @[ tba_started$$, true ]
           @[ X1$$, true ]
           @[ z0$$, 0 ]
           @[ z1$$, 0 ]
       NDESAC
   DESAC
   DCASE
     [ ] tba_started$$ :
       DCASE
         [ ] z0 < 11 :
           @[ z0$$, z0 + 1 ]
       DESAC
       DCASE
         [ ] z1 < 4 :
           @[ z1$$, z1 + 1 ]
       DESAC
       @[ failure$$, true ]
       NDCASE
         [ ] (X3$$) and (not switch2red and no_red_err )
                                             and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X3$$, false ]
           @[ X3$$, true ]
         [ ] (X3$$) and (not no_red_err ) and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X3$$, false ]
           @[ X5$$, true ]
         [ ] (X3$$) and (switch2red and no_red_err) and
            (z0$$ <= 10) and (z1$$ >= 2) and (z1$$ <= 3 ) :
           @[ failure$$, false ]
           @[ X3$$, false ]
           @[ X4$$, true ]
         [ ] (X5$$) and (TRUE ) and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X5$$, false ]
           @[ X5$$, true ]
         [ ] (X4$$) and (lights_on and no_red_err)
                            and (z0$$ >= 10) and (z0$$ <= 10 ) :
           @[ failure$$, false ]
           @[ X4$$, false ]
           @[ X5$$, true ]
         [ ] (X4$$) and (not no_red_err ) and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X4$$, false ]
           @[ X5$$, true ]
```

Figure A.4: SMI property translation(1)

```
         [] (X4$$) and (not lights_on and no_red_err )
                                             and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X4$$, false ]
           @[ X4$$, true ]
         [] (X1$$) and (not no_red_err ) and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X1$$, false ]
           @[ X5$$, true ]
         [] (X1$$) and (not ack and not turn_on and
                              no_red_err ) and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X1$$, false ]
           @[ X1$$, true ]
         [] (X1$$) and (ack and turn_on and no_red_err )
                                             and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X1$$, false ]
           @[ X2$$, true ]
           @[ z0$$, 0 ]
         [] (X2$$) and (switch2yellow and no_red_err)
                                            and (z0$$ <= 10 ) :
           @[ failure$$, false ]
           @[ X2$$, false ]
           @[ X3$$, true ]
           @[ z1$$, 0 ]
         [] (X2$$) and (not no_red_err ) and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X2$$, false ]
           @[ X5$$, true ]
         [] (X2$$) and (not switch2yellow and no_red_err )
                                             and (z0$$ <= 10) :
           @[ failure$$, false ]
           @[ X2$$, false ]
           @[ X2$$, true ]
       NDESAC
       @[ fairness_observer$$, false ]
       DCASE
         [] X5 :
           @[ fairness_observer$$, true ]
       DESAC
       DCASE
        [] failure$$ = true :
           @[ X1$$, false ]
           @[ X2$$, false ]
           @[ X3$$, false ]
           @[ X4$$, false ]
           @[ X5$$, false ]
           @[ non_failure acceptance$$, false ]
       DESAC
   DESAC
 OD
END
```

Figure A.5: SMI property translation(2)

Appendix B

Statemate model certifier patterns library

The STATEMATE model certifier helps the user to construct complex specifications simply by instancing or combining patterns. The present description is based on [I-L00a].

A pattern consists of a main part, called the *kernel pattern*, which defines the main property to be checked. For example, `(System_mode = RUNNING) implies (out1 = 1) after two steps`. This pattern is of the form *Condition implies Condition after Step-Count.*

We use the letters `P`, `Q`, `R` as place holders for conditions and `X`, `Y`, `N` for counters. Hence the property above is based on the kernel pattern called `P_implies_Q_X_steps_later`. Using this pattern the placeholders `P` and `Q` are respectively instantiated with `System_mode = RUNNING` and `out1 = 1` and `X` is instantiated with 2.

Besides the kernel part a pattern is further determined by its *mode* and its *start-up phase.*

The *start-up phase* can be defined either by giving a fixed number of steps or a predicate (e.g. raising a signal). Often the start-up phase is given by one single step where all initializations of the system are done. In STATEMATE this may correspond to the execution of the initial default transition.

The *mode* specifies when the kernel pattern should be valid. We distinguish four modes: initial, first, invariant, and iterative. Three of them were already defined in section 1.5, all are illustrated in figure B.1.

- *Initial* patterns define properties which should be valid for the initial part of a computation, i.e. when reaching a stable state after the start-up phase.
- The *First* pattern is an implication where the second part should be valid when the first part is true for the first time after the start-up phase.
- *Invariant* patterns should be valid again and again. Whenever the activation condition of a pattern is satisfied the specification of a pattern should be valid from that point onwards.

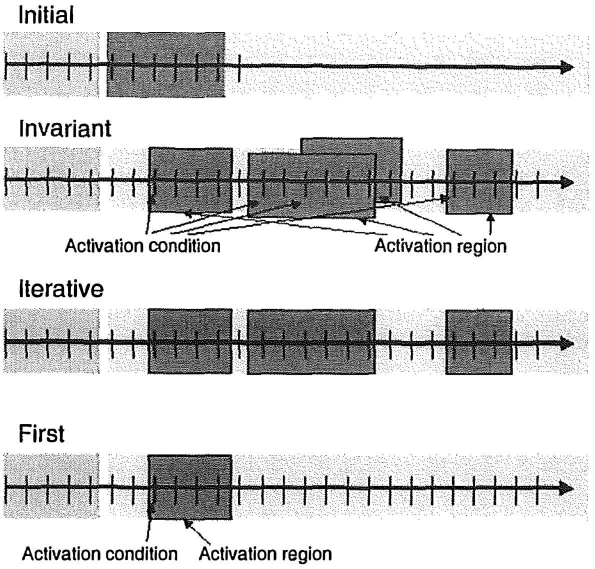


Figure B.1: The STATEMATE activation modes

- *Iterative* mode is similar to the invariant pattern in the sense that the pattern should be valid again and again. The difference with regards to the previous one is that this pattern will only be activated if the previous active region has been completed.

All the patterns

All the patterns available in the STATEMATE model certifier library are presented in table B.1, based on their kernel pattern and activation mode. A X means this patterns exists in this activation mode, a 0 states this combination does not exists, a 1 means this combination results in the same pattern as the one in initial mode and a 2 is the same as in invariant mode.

One should add the suffix _after_N_steps or _after_reaching_R to these kernel patterns to express the start-up-phase, hence obtaining the complete pattern. We notice that a final _B in the kernel pattern is a (finite) bound on the occurrence of the *finally* operator.

We can for instance build the complete pattern init_P_implies_finally_Q_B_after_reaching_R using an initial activation mode and a start-up phase triggered by the rise of a signal. The automaton corresponding to this property can be found in figure B.2.

All the patterns cited above are expressed in both TBA and timed temporal logics formalisms in [I-L00a].

Pattern mode Kernel property	initial	first	invariant	iterative
P	X	1	X	2
P_implies_finally_Q_B	X	X	X	2
Finally_P_B	X	1	X	2
P_implies_finally_globally_Q_B	X	X	X	2
Finally_globally_P_B	X	1	1	2
P_implies_globally_Q	X	X	X	2
P_implies_Q_X_steps_later	X	X	0	X
P_implies_Q_during_next_X_steps	X	X	0	X
P_implies_Q_atleast_X_steps_after_P	X	X	0	X
P_stable_X_steps_implies_afterwards_Q	X	X	0	X
P_stable_X_steps_implies_finally_Q_B	X	X	0	X
Q_while_P	X	1	X	2
Q_while_P_B	X	1	X	2
Q_only after_P	X	1	0	2
Q_not before_P	X	1	0	2

Table B.1: patterns of the STATEMATE model certifier library

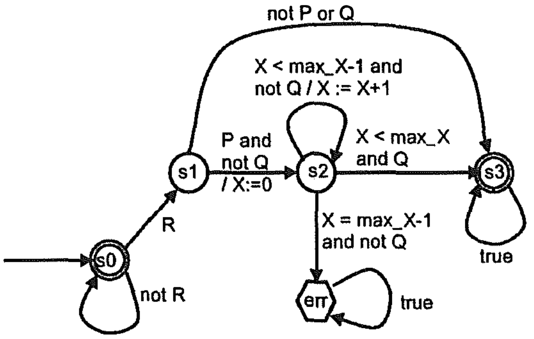


Figure B.2: The automaton of `init_P_implies_finally_Q_B_after_reaching_R`