UNIVERSITY JOSEPH FOURIER – GRENOBLE 1

THESIS

To obtain the grade of

UJF DOCTOR

Speciality: Mathematics and Computer Science

Presented and Defended in Public

by

Dejan NIČKOVIĆ

on October, 29th 2008

CHECKING TIMED AND HYBRID PROPERTIES: THEORY AND APPLICATIONS

Prepared in the **Verimag** laboratory

under the supervision of

Oded MALER

Jury

Saddek BENSALEM	President
Rajeev ALUR	Reviewer
Philippe SCHNOEBELEN	Reviewer
Eugène ASARIN	Examinator
Thomas HENZINGER	Examinator
Kevin JONES	Examinator
Oded MALER	Director

October 2008

Ćopkici

Acknowledgements

Doing my PhD thesis in Verimag has been an amazing experience, and I am fully indebted to my supervisor Oded Maler for . . ., well, a lot of things that are difficult to summarize in few words. He introduced me to the world of formal methods and verification and taught me almost everything that I know in the field. He was fully supportive, understanding, involved and present during all the phases of my PhD research and generously shared his views and experience on science, world and life. Thank you Oded for being much more than a thesis director to me and many kisses to Dorit, Mihal and Ouri for their gentleness and warm hospitality.

I am particularly grateful for having had the opportunity to collaborate with Amir Pnueli during my thesis. This thesis was also possible thanks to this collaboration and Amir's wisdom, elegance of thought and extreme kindness were always very inspiring to me. I would like to specially thank Eugène Asarin for many stimulating conversations and advices. From the Verimag laboratory, there is a number researchers who helped me understand different problems and influenced my research throughout scientific (and less scientific) discussions. I would like to acknowledge them, especially Marius (for his involvement and patience about IF questions), Thao, Saddek, Stavros, Paulo and Yassine, just to mention few of them. I would like to thank in particular Joseph Sifakis and Nicolas Halbwachs for continuously providing their support at different stages of my thesis.

Special thanks go to Kevin Jones, Victor Konrad and the rest of their group for giving me the opportunity to do an internship in Rambus and introducing me to analog circuit validation. They made me feel part of their team from the very beginning and made my journey in California a very pleasant experience. I also thank Marko for his friendship (and all the coffee breaks), as well as Matteo, Marco, Anna, . . .

The Grenoble years would not have been the same without the support and love of all my friends who made life much more than just research. So many thanks go to kum Odyss + kuma Ana, Radu, Goran and Alex, Jasmina, Maria and Christina, Julien, Selma, Noa, Alessandra and all the others.

Finally, this thesis could not have been done without the unconditional love from my parents Slobodanka and Slobodan and my brother Bojan who always gave me their full support in life. I dedicate this thesis to my mom, who deeply influenced and shaped the person that I am today. She was so excited and enthusiastic about my thesis, but left us too early to see it achieved.

Abstract

The growth of consumer *embedded devices*, where digital, analog and software components are often combined together on a single chip, results in an increase of complexity of the design and verification processes. The validation of such analog and mixed-signal systems largely relies on simulation-based techniques combined with often ad-hoc analysis methods. This thesis is motivated by the export of property-based formal techniques to the validation of analog and mixed-signal systems, at their continuous and timed levels of abstraction.

Since the formal verification of non-trivial continuous systems remains very difficult, we resort to a lighter validation technique, that is, *property-based monitoring*. We define *signal temporal logic* STL as a high-level specification language that allows expressing temporal properties of continuous and timed signals. STL is as an extension of the real-time *metric interval temporal logic* MITL, where continuous signals are transformed into Boolean ones using numerical predicates, and the temporal relations between them are expressed using standard real-time temporal operators whose atomic propositions correspond to those predicates. We develop two monitoring procedures, *offline* and *incremental*, for checking the correctness of simulation traces with respect to STL properties and implement them into a stand alone *analog monitoring tool* (AMT). The property-based monitoring framework is applied, using the AMT tool, to two real-world case studies, considering properties of a FLASH memory cell and a DDR2 memory interface.

We also consider the problem of property-based formal verification of timed systems, and develop a modular translation from MITL formulae with past and future operators to *timed automata*. The construction that we propose is based on *temporal testers*, a special class of input/output timed automata that realize the sequential functions defined by the semantics of MITL operators. We first show how every MITL formula can be expressed using six basic temporal operators (three for past and three for future) and show how to build a temporal tester for each of these operators. Temporal testers for arbitrary MITL formulae are obtained by composing these elementary testers.

Finally, we develop a procedure for automatic *synthesis* of *controllers* from high-level specifications expressed in the *bounded* fragment of *metric temporal logic* (MTL). We propose a translation from properties specified in this real-time logic and under *bounded variability* assumption, into *deterministic* timed automata to which we apply safety synthesis algorithms to build a controller that satisfies the specification by construction.

Résumé

Le développement croissant de *systèmes embarqués* de consommation, où les composants numériques, analogiques et logiciels sont combinés sur une même puce, résulte en une augmentation de la complexité des processus de conception et de vérification. La validation de tels systèmes analogiques et à signaux-mixtes reste largement basée sur des techniques de simulation, qui sont souvent combinées avec des méthodes d'analyse de nature ad-hoc. Cette thèse est motivée par l'exportation de méthodes formelles basées sur des propriétés, vers leur application à la validation de systèmes analogiques et à signaux mixtes, considérés à leur niveaux d'abstraction continu et temporisé.

Etant-donné que la vérification formelle de systèmes continus non-triviaux reste très difficile, nous nous tournons vers une méthode de validation plus légère appelée le *monitoring basé sur des propriétés*. Nous définissons *signal temporal logic* STL comme langage de spécification de haut niveau qui permet d'exprimer des propriétés temporelles de signaux continus et temporisés. STL est une extension de la logique de temps-réel *metric interval temporal logic* MITL, où les signaux continus sont transformés en signaux Booléens avec des prédicats numériques, et les relations temporelles entre ces signaux son exprimées avec les opérateurs temporels habituels dont les propositions atomiques correspondent à ces prédicats. Nous développons deux procédures de monitoring, une *offline* et une *incrémantale*, qui permettent de vérifier si les traces de simulations sont correctes par rapport aux propriétés STL. Les deux procédures sont implantées en *outil de monitoring analogique* AMT. Notre approche de monitoring basé sur des propriétés est appliquée, en utilisant AMT, à deux études de cas réalistes, où nous étudions des propriétés d'une mémoire de type FLASH et d'une interface de mémoire DDR2.

Nous considérons aussi le problème de vérification formelle de systèmes temporisés, et développons une traduction modulaire des formules MITL avec les opérateurs futurs et passés, vers des *automates temporisés*. La construction que nous proposons est basée sur les *testeurs temporels*, une classe spécifique d'automates avec les entrées et les sorties qui réalisent la fonction séquentielle définie par la sémantique des opérateurs MITL. Nous montrons d'abord comment chaque formule MITL peut être exprimée avec six opérateurs basiques (trois opérateurs passés et trois futurs) et nous proposons une construction de testeurs temporels à partir de ces opérateurs. Les testeurs temporels pour des formules MITL arbitraires sont obtenus en composant ces testeurs élémentaires.

Finalement, nous développons une procédure pour la *synthèse* automatique de *contrôleurs* à partir des spécifications de haut niveau exprimées avec le fragment *borné* de *metric temporal logic* (MTL). Nous proposons une traduction des propriétés spécifiées dans cette logique temporisée vers des automates temporisés *déterministes*, en supposant la *variabilité bornée*. Ensuite, nous pouvons appliquer à ces automates les algorithmes habituels de synthèse de sûreté pour construire un contrôleur qui satisfait la spécification par construction.

Contents

List of Figures

List of Tables

1 Introduction

The constant technological progress results in the design of increasingly complex systems that introduce richer functionality on smaller devices. The *electronic design automation* (EDA) industry provides various tools that aim to support engineers during different stages of the design flow. Despite this large palette of tools, the growing pressure to speed-up the production of high performance, low power and reliable devices makes the design process more vulnerable to faults. Such problems are often attacked using *formal verification* methods that allow detection (preferably during early stages) potential errors in the design.

In the context of digital hardware, the EDA tool support for engineers is mature and allows a high level of design automation. This progress has allowed huge scaling of digital designs over the past few decades. However, industrial estimations show that still about 70% of the overall design phase is dedicated to validation in its various forms. Consequently, different verification techniques for checking the correctness of a digital design have been studied extensively over the past decades, and successfully integrated into EDA toolkits. Formal verification procedures, such as *model* and *equivalence checking* or *theorem proving* aim at showing full system correctness. A lighter approach to verification, also called "dynamic" verification (monitoring) remains popular among the engineers, thanks to its relative simplicity with respect to the exhaustive checking framework. In this setting, the system is seen as a "black-box" that generates a finite set of behaviors that are checked against the specification for their correctness. Although incomplete, "dynamic" verification is effectively used to catch faults in the system, without guaranteeing its full correctness.

In recent years, the explosive growth of consumer embedded systems such as cell phones, GPS systems and portable multimedia devices resulted in "pushing" more technology on a single chip and combining together digital and *analog* components. The passage from purely digital to analog and mixed signal components is not trivial and adds another level of complexity to the design process. As a result, the potential of inserting an error into a design becomes higher, yielding an increasing need for automated analog and mixed-signal (AMS) verification tools. Validation of AMS designs in industry still relies mainly on simulation-based testing, combined with a number of common (and heterogeneous) analysis techniques, such as frequency-domain analysis, statistical measures, parameter extraction, eye diagrams etc. The tool support is usually specific to the class of properties considered and includes wave calculators, measuring commands as well as manually written scripts. These solutions are often ad-hoc and support minimal automation resulting in a time-consuming process that requires considerable (often non-reusable) user effort. The additional issue in AMS validation is the time required for the simulation of complex designs. A typical simulation of several nanoseconds of real-time transient behavior of a complex AMS circuit often takes hours or even days of simulation time (see Table 1.1). A number of recent articles [Dam08, Sei08, Sub07, Mau08] in specialized press urges for development of more automated tools to support the analog design flow, with a particular emphasis on AMS verification techniques. According to a small survey (see [Dam08]), 75% of analog designers questioned responded that they considered improved AMS verification tools and methodology as the greatest single need for enhanced EDA solutions in the AMS design[1].

Circuit	Simulation time
Driver	2.3h
802.11 #1	2.3h
Σ/Δ ADC	3.3h
DDR2	24.0h
I/O	176.2h
CDR	336.0h

Table 1.1. Simulation time for several AMS circuits [Sub07]

The general motivation of this thesis is to study different methods for extending some ingredients of verification methodology from digital (discrete) to analog and mixed-signal (timed, continuous and hybrid) systems. We adopt a property-based approach to verification, in which the system behavior is checked with respect to a high-level specification written in a formal language. We first define in Section 1.1 a generic model of a dynamical system defined over an abstract state space which evolves in an abstract time domain. The particular classes of models that we use can be obtained as special instances of this model. Section 1.2 describes different levels of abstractions (discrete, timed and continuous/hybrid) at which we consider systems and Section 1.3 introduces in more details some of the main formal techniques for checking properties of system behaviors. In section 1.4 we present the contributions of the thesis and present some of the related work on that subject (section 1.5). We finally conclude the introduction with section 1.6 by describing the thesis structure.

1.1 Systems and Properties

Systems, independent of the level of abstraction at which they are considered, react to changes in their environment (inputs, etc.) and generate output traces that are observable

[1] Although this survey is informal and not significantly large to make definite conclusions, it gives an interesting insight in the current preoccupations of analog designers

by the user. The correctness of the system can be defined in terms of the relationship between input and output traces using a formal specification language.

States and Behaviors

A model S of a system is defined over a set $V = \{v_1, \ldots, v_n\}$ of *state variables* each ranging over a domain V_i. The *state space* of the system is thus $V = V_1 \times \ldots \times V_n$. The system evolves over a time domain $\mathbb{T}$ which is a linearly-ordered set. A *behavior* of the system is a function w from the time domain to the state space $w : \mathbb{T} \to V$. A behavior can be either *complete*, with w defined all over $\mathbb{T}$, or *partial*, where w is defined only on a downward-closed subset of $\mathbb{T}$, that is, some interval of the form $[0, r)$. We use the notation $w[t] = \bot$ when $t \geq r$. We denote the set of all possible (complete and partial) behaviors[2] over a set V by V^*.

Systems

The dynamics of a system S is defined via a rule of the form $v' = f(v, u)$ which determines the *future* state as a function of the *current* state and *current* input $u \in U$. For some systems, there is no access to f and the interaction with the model is restricted to stimulating it with an input sequence which is in U^* and then observing the generated behavior w and checking its correctness.

Properties

Regardless of the formalism used to express it, a property φ defines a subset $L(\varphi)$ of V^*. A property monitor is a device or an algorithm for deciding whether a given behavior w satisfies φ (denoted by $w \models \varphi$), or, equivalently, whether $w \in L(\varphi)$. The most popular formalisms used to express properties are either based on *temporal logic* or *regular expressions*.

1.2 Levels of Abstraction

Different systems are defined at different levels of abstraction, depending on their functionality and the behaviors that they generate. While a *synchronous digital circuit* evolves over discrete time steps called “cycles” and generating values that are Boolean (or other finite domain) vectors, an analog amplifier transforms continuous real-valued signals. We identify *discrete, timed* and *continuous/hybrid* systems as classes of systems of particular interest in the context of this thesis.

[2] For discrete-time behaviors, it is common to use V^* for finite behaviors and V^ω for infinite ones, but these distinctions are less meaningful when we come to continuous behaviors.

Discrete Systems

Digital systems, such as software or digital hardware described at gate level and above, are usually modeled using discrete models. At this level of abstraction the set $\mathbb{N}$ of natural numbers is taken as the underlying time domain. In this case the difference between $w[t]$ and $w[t+1]$ reflects the changes in state variables that occur in the system within one clock cycle (hardware) or one program step (software). The state space of digital systems is often viewed as the set $\mathbb{B}^n$ of Boolean $n-$bit vectors[3]. Behaviors are, hence, n-dimensional Boolean *sequences* generated by system models which are essentially finite-state transition systems (automata) which can be encoded in a variety of formalisms, such as systems of Boolean equations with primed variables or unit delays, hardware description languages at various levels of abstractions (such as VERILOG or VHDL), programming languages, etc.

Timed Systems

Timed systems are discrete systems that evolve over a *physical* time scale modeled by real numbers. This level of abstraction is useful when the system does not have a central clock that defines "cycles" or when considering time-dependent behaviors of digital systems (such as gate delay propagation and timing analysis). Mathematically speaking, the behaviors generated by a timed system are *Boolean signals*, that is functions from $\mathbb{R}_{\geq 0}$ to $\mathbb{B}^n$ rather than sequences from $\mathbb{N}$ to $\mathbb{B}^n$. *Timed automata* [AD94] are often used to model systems that evolve over dense time. They are finite-state automata augmented with auxiliary continuous variables called *clocks* that can measure time between different events.

Continuous and Hybrid Systems

The state variables of continuous systems range over subsets of the set of *real numbers* that, in the case of analog circuits, represent magnitudes such as voltage or current. When considering AMS circuits, there can be several modes in which the analog components operate, that is the continuous dynamics of analog components may change according to the particular (discrete) mode. The behaviors generated by a hybrid/continuous system are *(piecewise)-continuous signals*, that is, functions from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}^n$. Such systems can be modeled by various formalisms such as differential equations or *hybrid automata* [MMP92, Hen96].

Example

As an example we consider the NAND gate, a simple component that forms the basic block of many circuits. A NAND gate can be viewed at different levels of abstraction

[3] In software, as well as in high-level models of hardware, systems may include state variables ranging over larger domains such as bounded and unbounded numerical variables or dynamically-varying data structures such as queues, stacks and trees, but at least in the hardware context, those can be encoded by bit vectors.

In_1	In_2	Out
0	0	1
0	1	1
1	0	1
1	1	0

Table 1.2. Truth table for a NAND gate

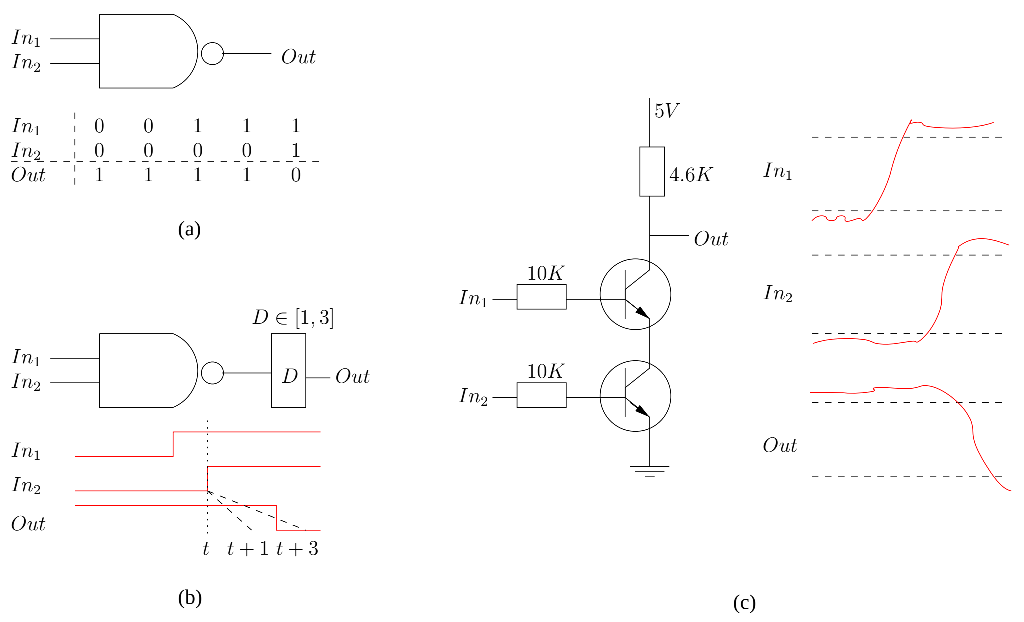


Fig. 1.1. A NAND gate at different levels of abstraction (a) discrete (b) timed (c) continuous

depending on the properties one wants to reason about. A NAND gate is usually represented at the digital level of abstraction, as show in Figure 1.1-(a). A single gate contains two input ports In_1 and In_2 and one output port Out. At each cycle, the NAND gate reads the current input values at In_1 and In_2 and, accordingly, generates the output value at Out. At this level of abstraction, one assumes that the NAND gate produces the output value according to the truth table (see Table 1.2) of the logical operator. The analysis of a network of NAND gates requires checking that the input/output logical behavior of the circuit corresponds to its specification.

While this abstraction is useful for reasoning about qualitative behavior of the circuit built from NAND gates, it assumes that the gates are perfect devices that generate the output at the end of the cycle upon reading the input values. In practice, that is not the case and a NAND gate takes some time to react to the inputs, processing the new values with a certain delay. When considering a network of NAND gates, individual delays are propagated throughout the circuit. One has to ensure that the delay propagation remains within acceptable limits and does not eventually affect the logical expected behavior of the circuit. In order to reason about such phenomena and do the *timing analysis* of the circuit, individual gates can be considered as timed systems and modeled using timed automata [Dil89, MP95b]. Figure 1.1-(b) represents a NAND gate with a non-deterministic delay that can range between 1 and 3 time units.

Finally, the actual physical implementation of the gates is done at the continuous level of abstraction using analog components such as transistors and resistors with real-valued state variables representing physical magnitudes (see Figure 1.1 (c)). From this level of abstraction, we can derive timing and power information for the component and analyze its correctness and robustness to variation of some (usually environment-controlled) parameters.

1.3 Checking Properties of System Behaviors

Formal Verification

Formal verification consists in proving the correctness of a system with respect to some formal specification. *Model checking* [CE81, QS82, BBF$^+$01, BK08, Dam08, HR04b] is a widely used algorithmic approach to verification where the entire state space of the underlying model of the system is explored. In that context, verification consists of checking whether *all* the (finite and infinite) behaviors generated by a system S satisfy its specification φ, that is effectively deciding the language inclusion $L(S) \subseteq L(\varphi)$. The general model checking framework is shown in Figure 1.2.

Efficient algorithms for model checking digital systems have been studied extensively over the past few decades. However, for very large systems, the exhaustive verification may still be intractable. Moreover, formal verification becomes in general much more difficult when you consider systems that are modeled with more details (such as systems that evolve in dense-time or have numerical/real variables).

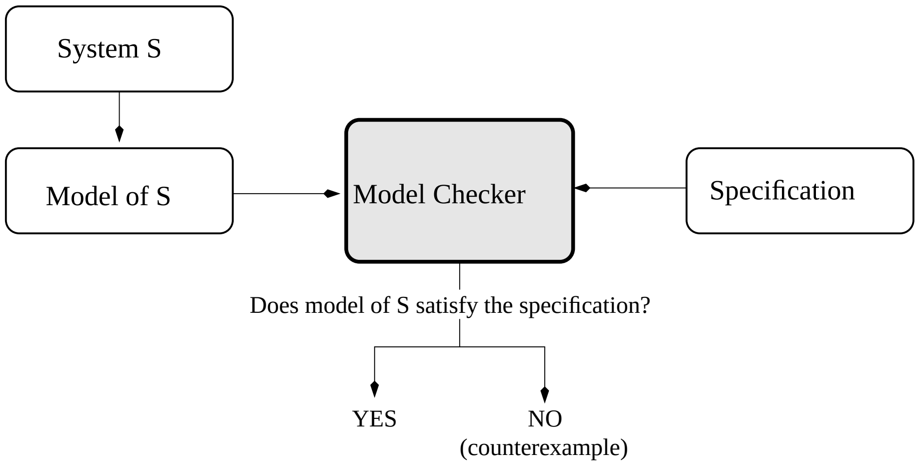


Fig. 1.2. Model checking

Monitoring

For systems which are outside the scope of automated verification tools, either due to the incorporation of unbounded variables (numbers, reals, queues) or simply due to their size or the lack of the underlying model, the preferred validation method remains based on testing/simulation. It has been noted that the formal specification component of verification can be still exported to simulation via the idea of *property monitors*. In the context of software, it is also known as *runtime verification*. Unlike the inclusion test $L(S) \subseteq L(\varphi)$ checked in verification, in monitoring one performs a *membership test* $w \in L(\varphi)$ on an *individual* behavior (simulation trace) w generated by the system S and the responsibility of the coverage is delegated to the test coverage generation procedure (or abandoned altogether).

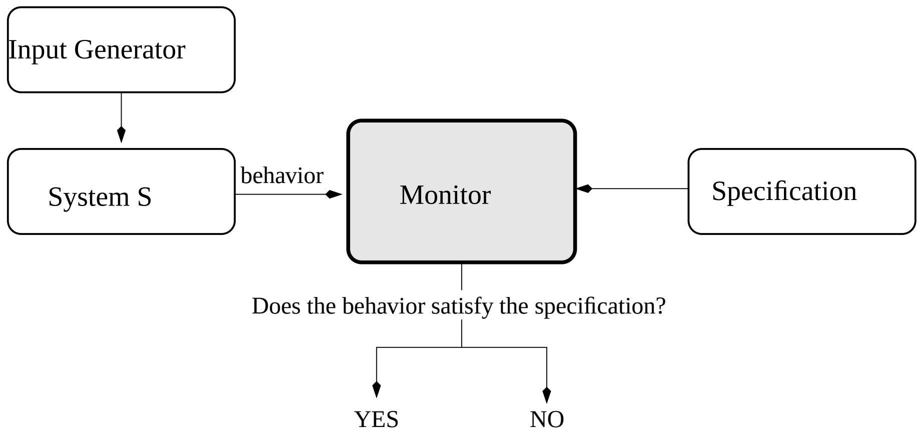


Fig. 1.3. Monitoring

Behaviors are generated by some kind of a *simulator* that computes states sequentially. They constitute the inputs for the monitor which checks whether they satisfy the property in question, as shown in Figure 1.3. Assuming that the simulator produces the behaviors forward (from past to future), one may think of three basic modes of interaction between the simulator and the monitor (see Figure 1.4):

1. **Offline:** The behaviors are completely generated by the simulator before the checking procedure starts. The behaviors are kept in a file which can be read by the monitor in either direction.
2. **Passive Online:** The simulator and the checker run in parallel, with the latter observing behaviors progressively as they are generated. This method allows early error detection and reporting the user as soon as a fault in a behavior is observed.
3. **Active Online:** There is a feed-back loop between the generator and the monitor where the latter may influence the choice of inputs and hence the subsequent values of the generated behavior. Such “adaptive” test generation may steer the system toward early detection of satisfaction or violation, and is outside of scope of this thesis.

There are some practical reasons to prefer one method over the other. First, to save time, we would like the checking procedure to reach the most refined conclusions as soon as possible. In the offline setting this will only reduce checking time, while in the online setting the effects of early detection of satisfaction/violation can be much more significant. This is because in certain systems (analog circuits represent a notorious example) simulation time is *very long* and if the monitor can abort a simulation once its satisfiability is decided, one can save a lot of time.

The difference between online and offline is, of course, much more significant in situations where monitoring is done with respect to a *physical device*, not its simulated model. We discuss briefly several instances of this situation. The first is when chips are tested after fabrication by injecting real signals to their ports and observing the outcome. Here, the response time of the tester is very important and early (online) detection of violation can have economic importance. In other circumstances we may be monitoring

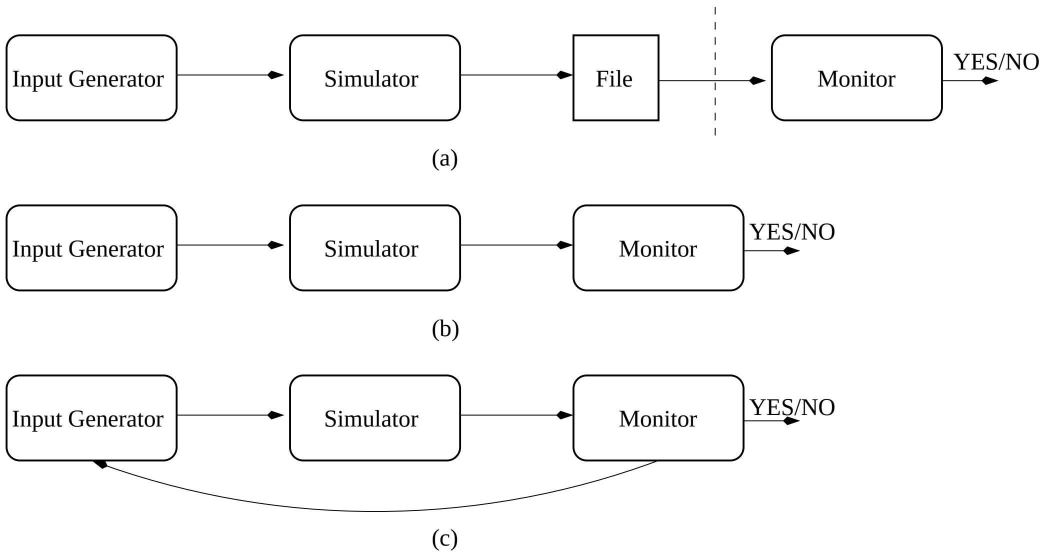


Fig. 1.4. Modes of interaction between a test generator and a monitor: (a) offline (b) passive online and (c) active online

a system which is already up and running. One may think of the supervision of a complex safety-critical plant where the monitoring software should alert the operator about dangerous developments that manifest themselves by property violation or by progress toward such violations. Such a situation calls for online monitoring, although offline monitoring can be used for "post mortem" analysis, for example, analyzing the "black box" after an airplane crash. Monitoring can be used for diagnosis and improvement of non-critical systems as well. For example analyzing whether the behavior of an organization satisfies some specifications concerning the business rules of the enterprise, e.g. "every request is treated within a week". Such an application of monitoring can be done offline by inspecting transaction logs in the enterprise data base.

Although the monitoring activity is *incomplete*, since it considers only a finite number of behaviors of the system, this lighter approach to verification presents some advantages when compared to its exhaustive counterpart:

- The system that is checked can be viewed as a black-box and its *model* is not needed (property checks are evaluated on simulation traces produced by the system, without the need to know *how* they were generated). This is an important feature when the model of the system is unknown or hardly formalizable (for example, even the simplest components in analog circuit design, such as transistors, are provided in form of closed libraries containing internally hundreds of differential and algebraic equations that model the component).
- Monitoring can be effectively used to catch errors in the system and report violations during the simulation process. When combined with some test coverage methods, monitoring the output traces can increase confidence in the system correctness.

- When the system is too large, the simulation-based analysis is the only tractable method to reason about it and monitoring provides a more systematic and rigorous approach to simulation/testing.

Synthesis

Formal verification and monitoring techniques aim at checking whether the behaviors generated by a *given* system S satisfy some high-level specification φ. Another approach, sometimes called *controller synthesis*, consists of starting from the specification φ, and generating automatically the system S that is guaranteed to be correct by construction (see Figure 1.5).

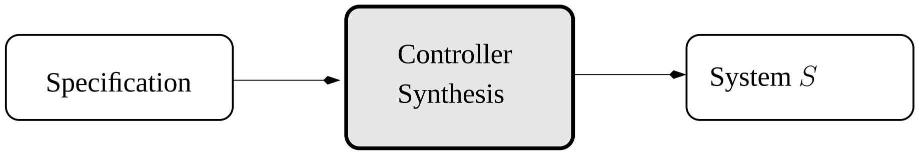


Fig. 1.5. Synthesis

The problem of synthesizing controllers automatically from high-level specifications can be stated as follows: given a property φ defined over two distinct action alphabets A and B (encoded using mutually-disjoint sets of variables), build a transducer (controller) S from A^ω to B^ω such that all of its behaviors satisfy φ.

1.4 Contributions of the Thesis

This thesis is motivated by the exportation of property-based formal techniques to the validation of timed and hybrid systems, mainly in the context of analog and mixed-signal circuits. Since the formal verification of non-trivial continuous and hybrid systems remains very difficult, we take a step forward by using an intermediate approach, that is *property-based monitoring*. We believe that the monitoring approach is appropriate for validation of analog and mixed-signal systems and is complementary to existing techniques that are already based on ad-hoc analysis of simulation traces. Following the observation that many interesting properties of transient simulation traces are expressed in the form of timing relations between signals, real-time extensions of temporal logics seem to form a solid basis for a property-based approach. In the context of purely timed systems, we are interested in methods for formal verification of real-time temporal logic properties. Finally, we also consider the problem of automatically synthesizing controllers (circuits) from real-time high-level specifications such that the controller satisfies the properties by construction. The contributions of this thesis can be summarized as follows:

1. In the context of analog and mixed-signal system validation, we created a comprehensive framework for monitoring properties of timed and continuous behaviors:
   a) We defined *signal temporal logic* STL as a high-level specification language for expressing properties of continuous and hybrid behaviors. STL is an extension of real-time *metric interval temporal logic* MITL [AFH96] where continuous signals are transformed into Boolean ones using a finite number of numerical predicates, and the temporal relations between them are expressed in a real-time temporal logic whose atomic propositions correspond to those predicates. These definitions are currently used as a basis for discussions toward the establishment of a new industrial standard.
   b) We developed two procedures for monitoring simulation traces against STL properties. The first one, first published in [MN04], is an *offline* procedure working on pre-existing simulation traces stored in a file. The second procedure is *incremental* and works in a piecewise-online manner to monitoring traces as soon as they are generated by the simulator. This procedure, first described in [MNP07b], can detect early violation/satisfaction of properties and reduce simulation time. The original algorithms in [MN04, MNP07b] were restricted to future temporal operators while those described in the thesis treat MITL in its full generality [AFH96] with both past and future temporal operators, as well as events.
   c) These monitoring procedures were implemented into a stand alone tool AMT (analog monitoring tool) first presented in [NM07]. In addition to the two monitoring procedures, AMT admits many features that help in defining properties, managing signals, visualization and interfacing with various simulators. The tool has been taken for evaluation by few semiconductor companies.
   d) The whole property-based monitoring methodology was applied, using the AMT tool, to two realistic case studies:

- Checking properties of FLASH memory cells as obtained from ST Microelectronics [NM07].
- Specifying timing properties from the official standard for DDR2 memory interface and checking them with respect to a set of simulation traces (in collaboration with Rambus, [JKN08]).

2. In the context of more formal verification (model checking) of timed systems, we developed a new modular translation from *metric interval temporal logic* MITL formulae to *timed automata*. Unlike the original translation of [AFH96], the construction that we propose is based on *temporal testers*, a special class of input/output timed automata (timed signal transducers) that realize the sequential functions defined by the semantics of MITL operators. An important advantage of this approach is that it requires the tester construction only for basic MITL temporal operators. Temporal testers for arbitrary MITL formulae are obtained just by composing the basic testers. Earlier versions of this translation were presented in [MNP05] for the *past* fragment of MITL and in [MNP06] for its *future* fragment. The version presented in this thesis is more complete, adhering to the full semantics of MITL as defined in [AFH96] providing a unified translation of MITL formulae with future, past and event operators to temporal testers. To the best of our knowledge this is the most direct translation from a real-time logic that can express past and events to timed automata.
   In addition to this contribution we believe that the construction provides a better understanding of real-time temporal logic. A prototype implementation of this construction into timed automata defined in the IF format has been developed.
3. We propose a complete chain for synthesizing controllers from high-level specifications. We consider the *bounded* fragment of *metric temporal logic* MTL [Koy90] as the specification language, and from real-time properties expressed in that logic we generate, under bounded-variability assumption, *deterministic* timed automata to which we apply safety synthesis algorithms to derive a controller that satisfies the properties by construction. This work was originally presented in [MNP07a].

1.5 Related Work

The need for system verification techniques has been addressed extensively by the formal methods community. In the context of digital systems such as hardware, a number of formal specification languages such as LTL or CTL have been proposed and studied, and an important part of research has been devoted to develop verification methods based on *model checking* [CE81, QS82, BK08, Dam08, HR04b] of such specifications. Temporal logic and regular expressions have been adopted as the basis for industrial specification languages PSL [HFE04] and SVA [Acc04] used in hardware industry and are currently supported by many commercial tools.

When considering *timed systems*, many variants of real-time temporal logics [Koy90, AH92a, Hen98, HR04a] as well as timed regular expressions [ACM02] have been proposed but the correspondence between simply-defined logics and variants of *timed automata* (automata with auxiliary clock variables [AD94]) is not as simple and canonical as for the untimed (digital) case, partly, of course, due to the additional complexity of the timed model. Consequently, existing verification tools for timed automata rarely use temporal properties. One of the most popular dense-time extensions of LTL is the logic MITL introduced in [AFH96] as a restriction of another real-time logic MTL [Koy90]. The decidability of MITL was established in [AFH96] and it was, together with MTL, subject to further investigations. However, model checking MITL properties [AFH96] remains complicated and, to the best of our knowledge, MITL has never been used in dense-time verification or monitoring tools. The only logic that has been integrated into a real-time model checking tool was the timed version of CTL, TCTL [HNSY94], used in the tool Kronos [Yov97].

In the context of monitoring properties of timed systems, a number of tools have considered integrating some restricted versions of real-time temporal logics. TemporalRover [Dru00] allows formulae in the discrete time fragment of the temporal logic MTL. TimeChecker [KPA03] is a real-time monitoring system with properties written in LTL$_t$ which uses a *freeze quantifier* to specify time constraints. The time notion in TimeChecker is discrete, but the monitoring steps are not done at the chosen resolution but are rather event-based. Another monitoring method based on temporal specifications expressed in MTL was presented in [TR04]. Their procedure can be seen as an event-based on-the-fly adaptation of tableau construction. The complexity of model-checking formulae of MTL, MITL and TCTL over restricted sets of timed paths was studied in [MR05]. In [BBKT04], the authors propose an automatic generation of real-time observers from timed automata specifications. They use a method of state-estimation to check whether an observed timed trace satisfies the specified property. This technique corresponds to an on-the-fly determinization of the timed automaton by computing all the possibles states that can be reached by the timed trace. No logic is used in that work.

Another, more ambitious extension of formal verification techniques involves systems that have continuous dynamics with switches, also called *hybrid systems*. The main direction involves studying *hybrid automata* [MMP92, Hen96], a mathematical model that allows to describe systems that have *continuous behavior* with mode switching, and developing algorithms for the exhaustive exploration of their state space. While hybrid automata are particularly well-suited to model AMS systems and progress is continuously being made in that field [ADF$^+$06], the bottleneck remains the exhaustive verification of their behavior, which still does not scale-up well, and can be currently applied only to small (often toy) examples. The relative lack of specification formalisms similar to LTL, but adapted to reason about hybrid systems results in only few property-based verification methods for hybrid automata [FGP06].

Recently, there have been several attempts to apply property-based monitoring procedures to continuous and hybrid systems. The authors of [JHP$^+$07] describe a framework based on PSL extended with analog operators, which is targeted at checking mixed signal interface properties. A similar approach for checking PSL properties of discrete time analog and mixed signals was proposed in [AZDT07]. In [DC05], the authors introduce an analog extension of CTL which they use to check properties of a finite state machine which represents a set of discretized and bounded transient simulation traces. The main limitation of these approaches compared to our framework, is that they all use discrete time as their underlying time domain.

1.6 Structure of the Thesis

2. **Temporal Logic on Discrete Behaviors:** this chapter introduces *temporal logic* as the high-level formal language for specification of digital systems properties with special emphasis on *linear temporal logic* (LTL) with *future* and *past* operators. The definition of LTL is followed by a discussion on its interpretation over incomplete (finite) behaviors in the context of monitoring. We present some common approaches for translating LTL properties into automata and describe an alternative translation based on a network of input/output automata called *temporal testers*. Our translation from MITL to timed automata is an extension of this construction to dense time.
3. **Timed Systems Preliminaries:** in this chapter, we present the basics of timed systems. First, we introduce dense-time Boolean *signals* as the semantic domain for timed systems. Then, we present the real-time temporal logic MITL which allows us to specify quantitative properties of timed systems. We also prove some basic properties of MITL which are used later in the procedures for monitoring and for translation into timed automata. Finally, we define *timed signal transducers* as an input/output variant of timed automata, and that will be used as the basic building blocks for the translation of MITL formulae to timed automata.
4. **Monitoring Timed Behaviors:** this chapter describes algorithms for checking MITL properties on finite timed behaviors. We first consider an offline procedure that can be applied to already existing timed traces and then present an incremental version of this algorithm which can be applied for online monitoring of MITL properties.
5. **Monitoring Continuous Behaviors:** in this chapter we extend MITL into the *signal temporal logic* STL for expressing temporal properties of real-valued (continuous, analog) signals. We discuss some issues related to the generation and representation of such signals inside the computer and adapt the monitoring procedure to these signals.
6. **Analog Monitoring Tool:** This chapter describes the structure and different features of the AMT tool implementing the monitoring procedures presented in chapters 4 and 5.
7. **Case Studies:** in this chapter, we describe the FLASH memory cell and DDR2 memory interface case studies in which we applied our approach for specifying properties of continuous and hybrid behaviors and monitoring the correctness of analog and mixed-signal simulation traces using the AMT tool.
8. **From MITL to Timed Automata:** We describe the construction of timed temporal testers for the basic MITL operators and thus, via composition, we build timed testers for arbitrary MITL formulae.
9. **On Synthesizing Controllers from Bounded-Response Properties:** in appendix A we present a procedure for synthesizing controllers from the bounded fragment of the MTL logic under bounded variability assumption. Since in this work the definitions of signals and of the logic differ from the rest of the document, we present the results in the form of the originally published paper [MNP07a].

2

Temporal Logic on Discrete Behaviors

Temporal logic is a rigorous formalism for specifying behaviors of discrete systems. It provides simple constructs to describe the order in which different "events" in the system should happen. Decision procedures for model-checking of temporal logic formulae [MP91, MP95a] play a central role in algorithmic verification of discrete transition system. In the linear-time context one takes the negation $\neg\varphi$ of the specification and derives from it an automaton-like device $\mathcal{A}_{\neg\varphi}$ that accepts exactly sequences of states that violate φ [VW86] and then checks whether the set of behaviors generated by the system model intersects the language of $\mathcal{A}_{\neg\varphi}$. For discrete-time models, used for functional verification of software or synchronous hardware, the logical situation is rather mature. Logics such as LTL (linear-time temporal logic) or CTL (computation-tree logic) are commonly accepted and incorporated into verification tools. For LTL a variety of efficient algorithms for translating a formula into an equivalent automaton have been proposed [GPVW95, SB00, GO01, KP05] and it even underlies industrial standards such as PSL [HFE04] and, to some extent, SVA [Acc04].

Temporal logic has been also used as the (bases for the underlying) specification language in a number of monitoring tools, including Temporal Rover (TR) [Dru00], FoCs [ABG$^+$00], Java PathExplorer (JPaX) [HR01] and MaCS [KLS$^+$02]. TR is a commercial tool that allows one to annotate programs with temporal logic formulae and then monitor them. FoCs is a monitoring system developed at IBM that automatically transforms PSL properties into checkers in the form of simulation blocks compatible with various HDL simulators. JPaX is a software-oriented runtime verification system for data race analysis, deadlock detection and temporal specifications. MaCS is another software-oriented framework aimed at runtime checking (and steering) of real-time programs. Unlike verification, where the availability of the system model allows one to reason about infinite sequences (carried by cycles in the transition graph), monitoring is usually restricted to finite-length behaviors, which often requires adapting the interpretation of temporal logic in some way. In [MS03], the authors show that the problem of checking whether a finite or ultimately periodic path satisfies a temporal logic formula can be usually solved efficiently.

When a temporal logic such as LTL is used in practice, one usually considers only its *future* fragment, where the temporal modalities refer to future occurrences of events. It has been argued that such a "futuristic" specification style is more natural for humans, and this approach has been indeed adopted by both industrial specification languages PSL and SVA. Moreover, the *past* fragment of LTL does not add any expressive power to its future fragment, when interpreted over sequences that have a starting point[1]. However, some properties can be expressed more naturally and succinctly by combining both past and future LTL operators. For example, the property "every p should have been preceded by a q", can be naturally expressed as [2] $\square(p \rightarrow \diamondminus q)$. In fact, it has been shown in [LMS02] that temporal logic with past can be exponentially more succinct than its pure future fragment. Another property (whose realization in dense time will be discussed in the sequel) is *rise(p)* which holds at time instants where p *becomes* true, can be naturally expressed as $p \wedge \neg \ominus p$, namely p *and previously not* p. A more exhaustive list of mixed future-past properties can be found in [KVR83].

In section 2.1 we define syntax and semantics of linear-time temporal logic LTL with both future and past fragments. The problem of interpretation of LTL over finite traces is discussed in section 2.2. In section 2.3, we describe some standard methods for translating LTL formulae into Büchi automata and describe in particular such a translation based on temporal testers.

[1] In other words, when the time domain is isomorphic to $\mathbb{N}$, rather than $\mathbb{Z}$. Languages over bi-infinite sequences have been studied in [NP86]

[2] Always p implies once in the past q. LTL operators are formally defined in 2.1.

2.1 Linear-Time Temporal Logic - LTL

Linear-time temporal logic (LTL) with *future* and *past* is defined using the following syntax:

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc \varphi \mid \ominus \varphi \mid \varphi_1 \mathcal{U} \varphi_2 \mid \varphi_1 \mathcal{S} \varphi_2$$

where p belongs to a set $P = \{p_1, \ldots, p_n\}$ of propositions. LTL is interpreted over n-dimensional Boolean ω-sequences of the form $w : \mathbb{N} \rightarrow \mathbb{B}^n$. We use $w[t]$ to denote the value of a sequence w at position t and abuse p to denote the projection of w on variable p. The semantics of LTL formulae is typically given via a doubly-recursive[3] definition of the relation $(w,t) \models \varphi$ indicating that a sequence w satisfies φ at position t, with the following rules:

$$\begin{array}{llcl}
p & (w,t) \models p & \leftrightarrow & p[t] = 1 \\
\text{not } p & (w,t) \models \neg\varphi & \leftrightarrow & (w,t) \not\models \varphi \\
\varphi_1 \text{ or } \varphi_2 & (w,t) \models \varphi_1 \vee \varphi_2 & \leftrightarrow & (w,t) \models \varphi_1 \text{ or } (w,t) \models \varphi_2 \\
\text{next } \varphi & (w,t) \models \bigcirc \varphi & \leftrightarrow & (w,t+1) \models \varphi \\
\text{previously } \varphi & (w,t) \models \ominus \varphi & \leftrightarrow & t > 0 \text{ and } (w,t-1) \models \varphi \\
\varphi_1 \text{ until } \varphi_2 & (w,t) \models \varphi_1 \mathcal{U} \varphi_2 & \leftrightarrow & \exists\, t' \in [t,\infty)\ (w,t') \models \varphi_2 \text{ and} \\
 & & & \forall\, t'' \in [t,t')\ (w,t'') \models \varphi_1 \\
\varphi_1 \text{ since } \varphi_2 & (w,t) \models \varphi_1 \mathcal{S} \varphi_2 & \leftrightarrow & \exists\, t' \in [0,t]\ (w,t') \models \varphi_2 \text{ and} \\
 & & & \forall\, t'' \in (t',t]\ (w,t'') \models \varphi_1
\end{array} \tag{2.1}$$

Basic LTL operators can be used to derive other standard Boolean and temporal operators, and in particular *eventually*, *always*, *once* and *historically* operators [4]

$$\begin{array}{ll}
\Diamond \varphi = \mathrm{T}\ \mathcal{U}\ \varphi & \diamondminus \varphi = \mathrm{T}\ \mathcal{S}\ \varphi \\
\Box \varphi = \neg \Diamond \neg\varphi & \boxminus \varphi = \neg \diamondminus \neg\varphi
\end{array}$$

The until formula $\varphi_1 \mathcal{U} \varphi_2$ requires that φ_2 will eventually occur. In some cases, a weaker property that requires that φ_1 holds continuously either until φ_2 occurs or throughout the whole duration of w is preferred to the standard *until*, and is expressed by *unless* operator $\varphi_1 \mathcal{W} \varphi_2$ which is equivalent to the formula $\varphi_1 \mathcal{U} \varphi_2 \ \vee\ \Box \varphi_1$. Similarly, one can define the *backto* operator $\varphi_1 \mathcal{B} \varphi_2$ which is the past equivalent of *unless* and can be expressed as $\varphi_1 \mathcal{S} \varphi_2 \ \vee\ \boxminus \varphi_1$.

[3] Both on the structure of the formula and on time.

[4] T stands for *true*.

2.2 Evaluation of LTL Formulae over Incomplete Behaviors

LTL was originally targeted at describing properties of *reactive systems* and the standard LTL semantics is defined over *complete infinite behaviors*. When considering the problem of monitoring, one does not exploit the model of the system S, but rather observes the behaviors of finite length that it generates. In this section, we discuss some problems related to the interpretation of LTL formulae over finite traces.

The satisfaction of a *past* LTL formula φ by a sequence w at any time t is determined according to the values of w at positions $t' \in [0, t]$ between the beginning of the trace and "now". In that sense, the definition of past LTL is *causal* and admits an immediate translation to deterministic automata and a simple monitoring procedure [HR02] based on this observation.

A major problem of monitoring properties expressed in the future fragment of LTL is due to the *acausal* definition of the satisfaction relation for temporal operators. In other words, the satisfiability of a formula φ at time t may depend on the value of the input sequence w at some *future* time $t' > t$. One of the questions is how to evaluate φ at the end of the trace, that is at a position from which we don't know what would happen in the future. After observing a finite sequence w, there are three possible basic situation with respect to its satisfaction of a property φ:

1. All possible infinite completions of w satisfy φ. Such a situation may happen, for example, when φ is $\Diamond\, p$ and p occurs in w. In this case we say that w *positively determines* φ.
2. All possible infinite completions of w violate φ, as in the case when φ is $\Box\, \neg p$ and p occurs in w. Then, we say that w *negatively determines* φ.
3. Some possible completions of w do satisfy φ and some others violate it. For example, any sequence where p has not occurred has extensions that satisfy, as well as violate, properties of the type $\Diamond\, p$ or $\Box\, \neg p$. In this case we say that w is *undecided*

This classification into positive, negative and undecided determination is tightly related to the characterization of LTL formulae into *safety* and *liveness* properties. A formula φ defines a safety property if and only if any sequence w that violates φ has a finite prefix that negatively determines φ. An example of such property is $\Box\, p$. A formula φ defines a liveness property if and only if for any finite word, there is an extension that satisfies φ. A typical liveness property is $\Diamond\, p$. Note that some formulae, such as $p \mathcal{U} q$ are neither, but can be decomposed into a conjunction of a safety and a liveness property, $p \mathcal{U} q = (p \mathcal{W} q) \wedge \Diamond\, q$.

The "undecided" category can be refined further according to methodological, quantitative and logical considerations. The quantitative aspects enter the picture as well because the longer we observe a sequence w free of p, the more we tend to believe in the satisfaction of $\Box\, \neg p$, although the doubt will always remain. On the other hand, the satisfaction of a formula like $\bigcirc^k p$, although undecided for sequences shorter than k, will be revealed within bounded time. The most general type of answer concerning the satisfiability of φ by a finite-length behavior w would be to give exactly the set of *completions* of w that will make it satisfy φ, defined as

$$w\backslash\varphi = \{w' : w \cdot w' \models \varphi\}.$$

Positive and negative determination correspond, respectively, to the special cases where $w\backslash\varphi = X^*$ and $w\backslash\varphi = \emptyset$. This "residual" language can be computed syntactically as the left quotient ("derivative") of φ by w.

In certain situations we would like to give a decisive answer at the end of the sequence. In case of positive and negative determination we can reply with a yes/no answer without ambiguity. For some sub-classes of LTL formulae an unambiguous finitary semantics (that guarantees positive/negative determination) can be achieved. The simplest among those is bounded LTL where the only future temporal operator is *next* $\bigcirc$ and where the satisfiability of a formula φ at time 0 is always determined by the values of the input sequence w up to some $t < k$, with k being a constant depending on φ. Note that this class is not useless as it might seem: one can use "syntactic sugar" operators such as $\Box_{[0,k]}\,\varphi$ as shorthand for $\bigwedge_{i=0}^{k}(\bigcirc^i\,\varphi)$. The implication for monitoring is that every *sufficiently-long* sequence is determined with respect such formulae (see also [KV01, MN04]).

Although useful, the class of bounded-LTL properties may not be sufficient. In some cases, the length of the finite behavior cannot be known in advance, and a-priori "bounding" of the property is not advised. Instead of specifying $\Box_{[0,k]}\,\varphi$ with a pre-defined bound k, a preferred solution would be to express the property as $\Box\,\varphi$, with the interpretation that φ has to hold continuously from time 0 until the end of the finite trace. This idea can be generalized, by interpreting any quantification over time $Qt, Q \in \{\forall, \exists\}$ as $Qt < |w|$ and hence a safety that has not been violated during the lifetime of w is considered as satisfied, and an eventuality not fulfilled by that time is interpreted as violated. This principle may be extended to more complex LTL formulae that involve nesting of temporal operators, although in this case the interpretation may seem less intuitive.

Naturally many solutions have been proposed to this problem in the context of monitoring and runtime verification and we mention few. The work of [ABG$^+$00] concerning the FoCs property checker of IBM, as well as those of [KLS$^+$02] are restricted to safety or eventuality properties and report violation when it occurs. On the other hand, the approach of giving the residual language is proposed in [KPA03] and [TR04] in the context of timed properties. The most systematic study of adapting LTL semantics to finite sequences ("truncated paths") is presented in [EFH$^+$03, EFH05], and has been adopted by the industry standard PSL.

The PSL language defines four levels of satisfaction of a property by a finite-length behavior, illustrated in Figure 2.2:

1. **Holds strongly:** the property has not been violated and all future obligation have been met. Moreover, the property is guaranteed to hold on every possible infinite completion of the behavior.
2. **Holds:** while the property has not yet been violated and all future obligations have been met, there are some possible completions of the behavior that satisfy, and other that violate the formula

3. **Pending:** the property has not been violated by the behavior, but not all of the obligations have been met by the finite trace. There are infinite completions that may or may not satisfy the formula
4. **Fails:** the property has been violated by the finite behavior and hence there is no extension of the behavior that will satisfy it

t	0	1	2	3
$w[t]$	$\overline{p}$	$\overline{p}$	$\overline{p}$	p

Fig. 2.1. Example of PSL levels of satisfaction wrt w: holds strongly for $\Diamond p$; holds for $\Box(\overline{p} \rightarrow \Diamond p)$; pending for $\Box(p \rightarrow \Diamond \overline{p})$; fails for $\Box \overline{p}$

The future fragment of LTL is part of the PSL language and its syntax and semantics are slightly adapted for being interpreted over both *finite* and *infinite* behaviors. The main extension with respect to standard LTL is the introduction of *strong* $\bigcirc^s$ and *weak* $\bigcirc^w$ next-time operators[5]. The distinction between these two operators is made only at the last position of the sequence. In fact, $\bigcirc^w p$ holds at the last position of the trace, while $\bigcirc^s p$ does not, independently of the input. The two versions of the next-time operator have the following semantics (see [EFH05]):

$$\begin{aligned}(w,t) &\models \bigcirc^s \varphi \leftrightarrow t < |w|-1 \text{ and } (w,t+1) \models \varphi\\ (w,t) &\models \bigcirc^w \varphi \leftrightarrow t \geq |w|-1 \text{ or } \ (w,t+1) \models \varphi\end{aligned}$$

Note that the weak version of the *eventually* operator $\Diamond^w \varphi$ and a strong version of the *always* operator $\Box^s \varphi$ do not make much sense when interpreted over a finite behavior. In fact, using expansion formulae, one can express the weak eventuality as $\Diamond^w \varphi = \varphi \vee \bigcirc^w \Diamond^w \varphi$ and it is clear that $\Diamond^w \varphi$ trivially holds for any finite behavior w. Similarly, $\Box^s \varphi$ is violated by any finite sequence.

[5] The same weak/strong distinction is also defined for boolean expressions because PSL can be interpreted over empty words

2.3 From LTL to Automata

The standard methodology for checking whether all the behaviors of a finite-state system S, modeled by an automaton $\mathcal{A}_S$, satisfy a specification expressed as a temporal property φ, involves building a Büchi automaton $\mathcal{A}_{\neg\varphi}$ that *accepts* exactly all the (infinite) words that violate the property φ. The model checking problem, that is, the language inclusion $L(\mathcal{A}_S) \subseteq L(\varphi)$ between the possible behaviors of $\mathcal{A}_S$ and the behaviors satisfying φ, reduces to the checking whether the product automaton $\mathcal{A}_S \times \mathcal{A}_{\neg\varphi}$ accepts the empty language, implying that there exists no computation of S which violates φ.

In the discrete-time domain, the construction of $\mathcal{A}_{\neg\varphi}$ typically follows a tableau-based procedure based on *expansion formulae* that separate the variable values that have to hold at the *current* position from the future obligations that are propagated to the *next* position, for example $\Box\,\varphi = \varphi \wedge \bigcirc\,\Box\,\varphi$. As one can see, the expansion rules rely heavily on the *next* operator $\bigcirc$ which allows to separate clearly current obligations from future ones. It is not hard to see that this idea cannot be applied in a straightforward manner to behaviors defined over a *dense* time domain.

The growing complexity of digital systems calls for more modular and compositional reasoning about them. Modern specification languages used in the EDA industry such as PSL [HFE04] or SVA [Acc04] adapted to this reality by providing constructs that facilitate specification of complex properties in a bottom-up fashion through the composition of lower-level component properties. On the other hand, traditional tableau-based acceptors are hard to adapt to this paradigm because they do not compose naturally. One reason for the lack of compositionality is that an acceptor $\mathcal{A}_\varphi$ provides information concerning the satisfaction of φ by the *entire* input sequence, that is, at position 0, but no information concerning satisfaction of φ at any position $t > 0$. Consequently, when $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$ are the acceptors constructed from formulae φ_1 and φ_2, respectively, there is no simple recipe to compose them to obtain an acceptor for the formula $\varphi_1 \mathcal{U} \varphi_2$. The property $\varphi_1 \mathcal{U} \varphi_2$ is satisfied iff there is a *future* position $t > 0$ where φ_2 is true, and that φ_1 holds continuously at all positions t' such that $0 < t' < t$. The acceptors $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$ do not provide this information.

An alternative style of construction (see [Var95]) uses *alternating automata* [CKS81], automata that employ both existential and universal non-determinism. The construction of alternating automata from formulae is, in some sense, more compositional and elegant as it works inductively on the structure of the formula, however it is not compositional in the following sense: the automaton for a formula may make *transitions* to the automata of its sub-formulae but it does not observe the evolution of their satisfiability over time. Moreover, since model-checkers deal only with existential non-determinism, the universal non-determinism has to be removed by a kind of subset construction [MH84] at exponential cost.

2.3.1 Temporal Testers

There exists a construction of automata from LTL formulae which is based on *temporal testers*, an orthogonal solution to the problem of compositionality where an additional structure imposes the responsibility of being composable on the automata for the sub formulae [KPR98, KP05]. Consider a simple formula φ consisting of one temporal or propositional operator defined over propositional variable $p_1, \ldots, p_n$. A temporal tester T_φ for φ is a *transducer* whose input alphabet is $\mathbb{B}^n$, the set of valuations of the propositional variables appearing in φ, and whose output alphabet is $\mathbb{B}$. While observing an input sequence w, the tester outputs a Boolean sequence u such that $u[t] = 1$ iff φ is satisfied at t, that is $(w, t) \models \varphi$. Hence, unlike an acceptor $\mathcal{A}_\varphi$ which tells us whether the entire input sequence satisfies φ, the temporal tester T_φ does so for *every suffix* of w. This additional structure allows testers to compose naturally: we can view the output of T_φ as a propositional variable u_φ satisfying $\Box(u_\varphi \leftrightarrow \varphi)$. For a formula φ which has φ_1 and φ_2 as sub-formulae we can then build a tester T_φ over input variables u_{φ_1} and u_{φ_2}, which amounts to taking the outputs of T_{φ_1} and T_{φ_2} as inputs for T_φ. A construction of such a network of testers for the formula $(p \wedge \bigcirc q)\ \mathcal{U}\ (\Box r)$ is illustrated in Figure 2.2. Below, we list some properties of temporal that make them particularly attractive:

1. The construction of temporal testers is completely modular. It suffices to build testers for basic temporal and logical operators, which in the case of LTL are basic operators $\bigcirc p$, $p\mathcal{U}q$, $p \wedge q$ and $\neg p$, where p and q are propositions. Testers for more complex formulae are constructed by composing these building blocks.
2. Temporal testers naturally support extensions of the specification language. Once a new language construct is introduced, its corresponding tester can be naturally composed with testers for existing operators. This feature has already been used to extend compositional construction of testers for LTL [KPR98] with the regular expression-like operators of PSL [PZ06a] and with branching-time operators of CTL* [KP05]. Likewise, the combination of future and past operators comes for free.
3. Testers for specific properties that have been expressed directly by an automaton or a program without a formal logical description, or that have been optimized [CRST06] can be combined with testers developed in a different way, as long as they produce the right output.
4. Unlike certain tableau-based techniques, the construction of temporal testers does not require the existence of expansion formulae. This is particularly important for testers defined algorithmically and for real-time logics such as MITL where the meaningfulness of the *next* operator $\bigcirc$ is not evident.
5. Although temporal testers are transducers that incorporate additional structure with respect to acceptors, the complexity of constructing such a tester for an arbitrary LTL formula is not worse than that of the acceptor. In its symbolic representation, the size of a tester is linear in the size of the formula. This implies that the worst-case state complexity is exponential for LTL and formulae, which is an established lower bound.

Temporal testers have several origins. To the best of our knowledge the idea of transducers that output the truth value of a temporal formula at each position was first proposed in [Mic84, Mic85] under the name *machines à formules* (*formulae machines*) as a way to reconcile logic-based and automaton-based approaches to semantics and verification. A similar idea has also been considered in [BCM$^+$92] in the context of symbolic implementation of a tableau construction. The observation that such a Boolean variable can replace the sub-formula itself in the context of model checking has been considered in [CGH94]. Surprisingly, these techniques did not get much attention in the verification community until recently. The properties of temporal testers have been studied in detail with respect to acceptors and alternating automata in [PZ06b] and much of the material in this chapter is based on it.

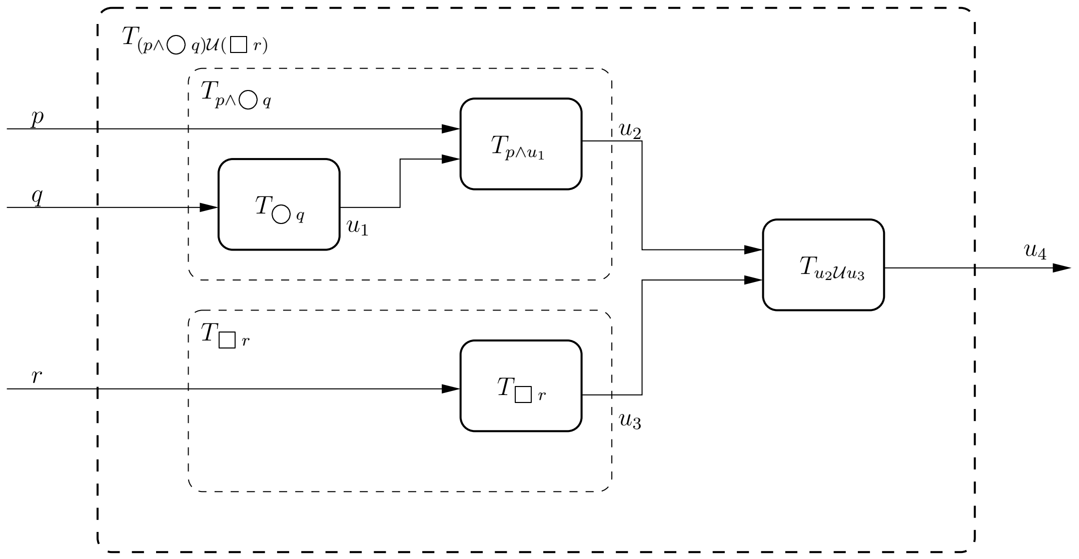


Fig. 2.2. Composition of temporal testers for $(p \wedge \bigcirc q)\ \mathcal{U}\ (\square\ r)$

2.3.2 Temporal Testers for LTL

In this section we show how to actually build testers for the basic LTL operators. We feel that, independently of its dense time generalization introduced in chapter 8, this construction, which makes use of *acausal transducers* as testers for the future temporal operators, may improve our understanding of temporal logic. We remind the reader that the satisfaction of a compound LTL formula $\text{OP}(\varphi_1, \varphi_2)$, where OP is a temporal or a propositional operator, by a sequence w at position t is an OP-dependent function of the satisfaction of the sub-formulae φ_1 and φ_2 by w at certain positions. The satisfaction relation can be viewed as *characteristic function* χ^φ which maps sequences over $\mathbb{B}^n$ into Boolean sequences such that $u_\varphi = \chi^\varphi(w)$ means that for every $t \geq 0$, $u_\varphi[t] = 1$ iff $(w, t) \models \varphi$.[6] Definition 2.1 can be seen then, as a recipe for building the characteristic function of φ from the characteristic functions of its sub-formulae, as illustrated in Figure 2.2. These characteristic functions which are to be realized by the temporal testers are instances of the class of *sequential functions* (transducers) which are functions that map sequences to sequences. A particular sub-class of sequential functions are the *causal* (sometime called *retrospective*) function.

Definition 2.1 (Causal Sequential Functions). *A sequential function $f : A^\omega \to B^\omega$ is said to be* causal *if for every $u \in A^*,\ v, v' \in B^*$ such that $|u| = |v| = |v'|$ and every $\alpha \in A^\omega$ and $\beta \in B^\omega$*

$$f(u \cdot \alpha) = v \cdot \beta \textit{ and } f(u \cdot \alpha') = v' \cdot \beta' \textit{ implies } v = v'$$

In other words, the value of $f(\alpha)$ at time t may depend only on the values $\{\alpha[t'] : t' \leq t\}$. Causal functions are realized naturally by *deterministic* automata with output (sequential synchronous transducers) that produce the next output symbol as they read the next input symbol. The semantics of the past fragment of LTL can be expressed using causal functions as the satisfaction of both *previously* $\ominus$ and *since* $\mathcal{S}$ operators *now* (at time t) is determined according to what have happened *until now* (positions $t' \leq t$).

The characteristic function of $\ominus\, p$ is nothing but a *shift* operator[7] defined by $u[t+1] = p[t]$. The temporal tester for $\ominus\, p$, shown in Figure 2.3-(a), is a simple one-bit input-driven shift register. Each time instant it reads the current value of p, memorizes it by going to the appropriate state and outputs the previous value as encoded by the source state of the transition.[8] Consider for example state s_0 where p is false and u is true (denoted by $\overline{p}/u$). When the next value of the input is p, the automaton will move to state $\overline{s}_1$ labeled by $p/\overline{u}$, while if it is $\overline{p}$ it will move $\overline{p}/\overline{u}$. The two states have the same output $\overline{u}$ which reflects the value $\overline{p}$ in the source state. Likewise transitions departing from p states $\{s_1, \overline{s}_1\}$ may

[6] We use u rather then of u_φ when φ is clear from the context. Recall that the relation between u_φ and φ can also be expressed by the formula $\Box(u_\varphi \leftrightarrow \varphi)$. We also use notations in the style p and $\overline{p}$ to denote 1 or 0 values of variable p.

[7] Known in other context as the delay operator z^{-1}.

[8] When outputs are associated with transitions, a one-place shift register has two states p and $\bar{p}$, but since we associate outputs with states as a preparation for timed automata over signals we split the state according to their output value u or $\bar{u}$.

only end up in u states $\{s_0, s_1\}$. The tester is input-deterministic, as from any state there is a single outgoing transition for a given input symbol.[9]

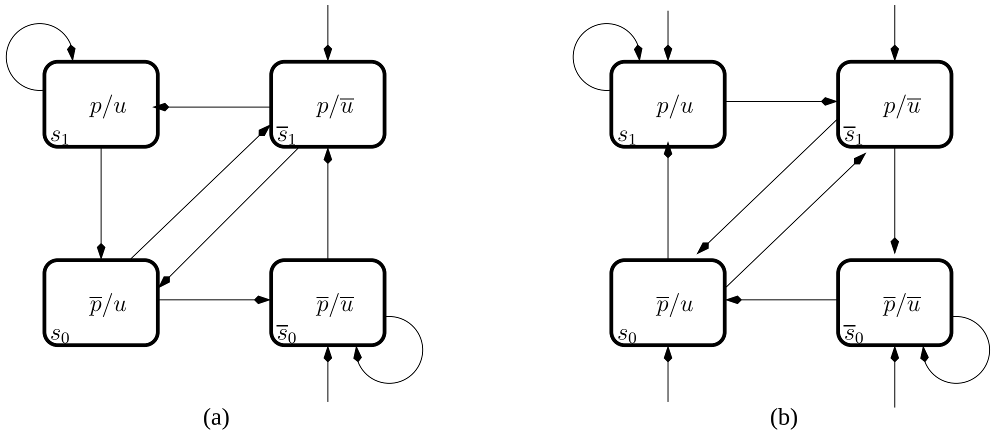


Fig. 2.3. Temporal testers for LTL formulae: (a) $\ominus\, p$; (b) $\bigcirc\, p$

On the other hand, the characteristic function associated with future LTL operators are not causal as the satisfaction at t may depend on satisfaction at some $t' > t$. The output of the *next* operator $\bigcirc$ at time t depends on the input at $t+1$ and, even worse, the output of the *until* operator $\mathcal{U}$ at t may depend on input values at arbitrary larger t'.

One can think of two ways to realize acausal sequential functions. The first approach, which works well for operators with a *bounded* level of acausality, for example $\bigcirc^d$ (the *next* operator nested d times), is to dissociate the time scales of the input and the output, that is, let the automaton ignore the first d input symbols, and then let $u[t] = p[t+d]$. Unfortunately, this does not work for unbounded acausality. In the alternative approach that we use, the temporal testers respond to the input *synchronously*, but since at time t the information might not be sufficient to determine the output, the tester has to "guess" the output non-deterministically and split the computation into two runs, one that *predicts* u and one that predicts $\overline{u}$. Each of the runs needs to remember the predictions it has made and, progressively, abort runs whose predictions turn out do be false. An automaton for an operator with acausality of depth d may need to memorize up to 2^d past predictions.

The similarity between remembering past observations (in a shift register) and remembering predictions is not a coincidence. The temporal tester for $\bigcirc\, p$, depicted in Figure 2.3-(b) can be obtained by *reversing* the transitions of the automaton for $\ominus\, p$. Note that, for finite sequences, if $u = \chi^{\bigcirc}(w)$ that $w^R = \chi^{\ominus}(u^R)$ where u^R and w^R are the reverses of the sequences u and w. The automaton for $\bigcirc\, p$ is output-deterministic and its state memorizes the prediction it made in the previous step and uses it to abort, in the next step, runs whose predictions turned out to be wrong.

[9] If we decide by convention that the output at the first instant is $\overline{u}$, we have $\overline{s}_0$ and $\overline{s}_1$ as initial states depending on the value of p.

To understand how such an acausal tester works let us look at Figure 2.4-(a) which shows the $\bigcirc\, p$-tester in an extended form where input/output labels appear also on transitions and where *abortions* due to wrong predictions are made explicit. States s_0 and s_1 indicate that the prediction made in the current step is u, hence from these two states, observing $\overline{p}$ contradicts the prediction and the run is aborted (*abort* transition). Input p confirms the prediction and the automaton splits the remaining run into two by moving non-deterministically to s_1 and $\overline{s}_1$ labeled by p/u and $p/\overline{u}$, respectively, thus generating two predictions for the next value and so on. For every ω-sequence w, only *one* infinite run survives and its output is $u = \chi^{\bigcirc p}(w)$. An initial prefix of a sample run is shown in Figure 2.4-(b).

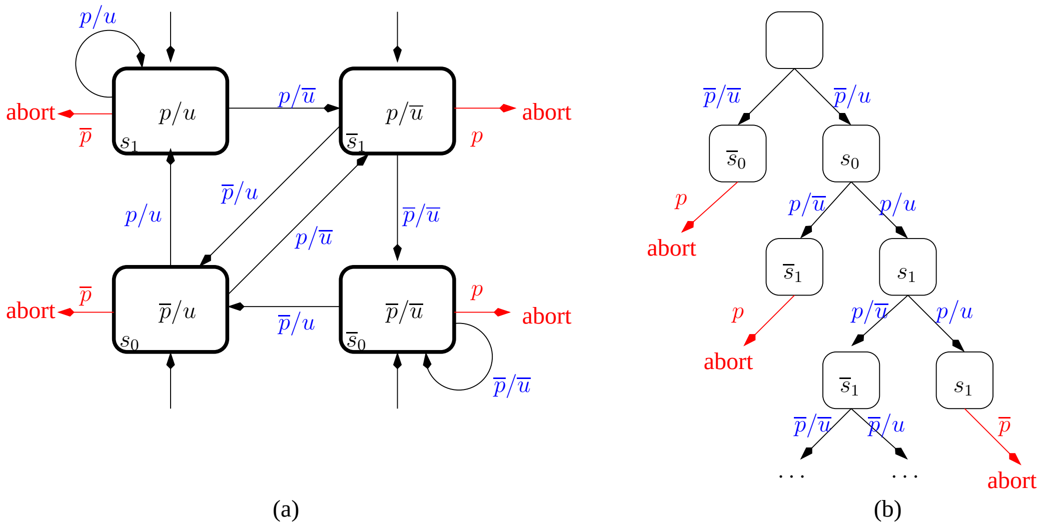


Fig. 2.4. Behavior of temporal testers: (a) the tester for $\bigcirc\, p$; (b) An initial fragment of the behavior of this tester for an input sequence $\overline{p}pp\overline{p}\cdots$ producing the output $uu\overline{u}\ldots$

The output of the past temporal tester for $p\mathcal{S}q$ is again fully determined by the observed past history. At positions where $\overline{pq}$ is observed, the property does not hold and the tester outputs $\overline{u}$. Likewise, when q is observed, the output is trivially determined to be u. In a state where $p\overline{q}$ is observed, the formula $p\mathcal{S}q$ can be either satisfied or falsified, depending on the previous observations, that is whether p has been continuously holding from the last time q was true. This situation is reflected by two states $s_{p\overline{q}}$ and $\overline{s}_{p\overline{q}}$, one outputting u and the other $\overline{u}$. In fact, the output is determined to be u, when $s_{p\overline{q}}$ state is entered from a q state and, likewise, the tester moves to state $\overline{s}_{p\overline{q}}$ when $p\overline{q}$ is observed right after observing $\overline{pq}$.

The situation with $p\mathcal{U}q$, although symmetric to $p\mathcal{S}q$, is more involved because a priori, due to the unbounded future horizon, one might need to generate and memorize 2^ω predictions. However, the semantics of *until* implies that at most *two* confirmable predictions may co-exist simultaneously.

Lemma 2.2. *Let* $u = \chi^{p\mathcal{U}q}(w)$. *Then for every* t *such that* $w[t] = w[t+1] = p\overline{q}$, $u[t] = u[t+1]$.

Proof. There are three possibilities: 1) The earliest $t' > t+1$ such that $w[t'] \neq p\overline{q}$ satisfies $w[t'] = q$. In that case, the property is satisfied both at t and $t+1$; 2) The same t' satisfies $w[t] = \overline{pq}$ and the property is violated both at t and $t+1$; 3) $w[t'] = p\overline{q}$ for every $t' > t+1$ and the property is falsified from both time points.[10]

This fact is reflected by the tester of Figure 2.5-(b). At time instants where $\overline{pq}$ is observed, the value of the output is determined to be $\overline{u}$. Likewise, when q is observed the output is determined to be u. The only situation that calls for non-determinism is when $p\overline{q}$ holds and we do not know in which of the three cases of Lemma 3.7 we will eventually found ourselves. Hence we split the run into positive and negative predictions (states $s_{p\overline{q}}$ and $\overline{s}_{p\overline{q}}$, respectively). The only input sequences that will lead to two infinite runs are those ending with $(p\overline{q})^\omega$. To choose the correct one among them we add a Büchi condition which requires that one of $\{\overline{s}_{\overline{pq}}, \overline{s}_{p\overline{q}}, s_q\}$ is visited infinitely often, which amounts to rejecting runs that stay in $s_{p\overline{q}}$ forever. With these four testers (and the trivial testers for the Boolean operators), one can build testers for arbitrary (past and future) LTL formulae.

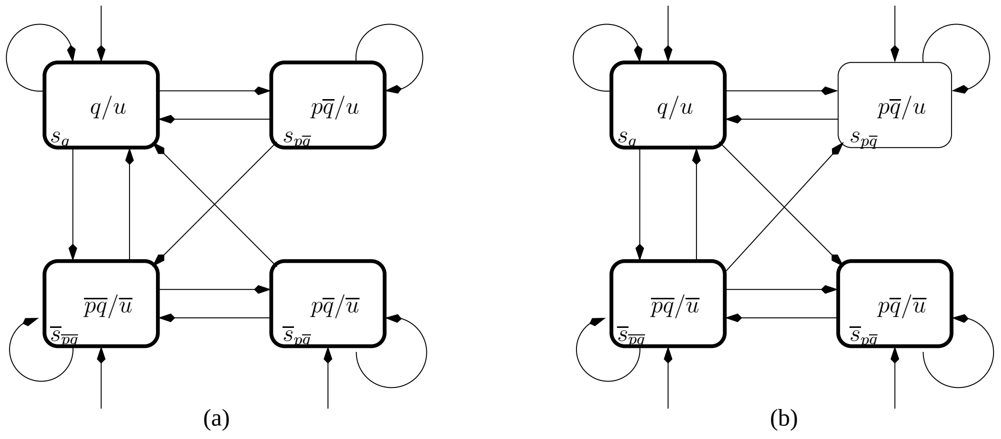

q/u
s_q
$p\overline{q}/u$
$s_{p\overline{q}}$
$\overline{pq}/\overline{u}$
$\overline{s}_{\overline{pq}}$
$p\overline{q}/\overline{u}$
$\overline{s}_{p\overline{q}}$
(a)
(b)

Fig. 2.5. Temporal testers for LTL formulae: (a) The tester for $p\,\mathcal{S}\,q$ tester; (b) The tester for $p\,\mathcal{U}\,q$. Accepting states are indicated by bold lines. Note that acceptance here has nothing to do with the satisfaction of the property but whether the sequential function $u = \chi^{p\,\mathcal{U}\,q}(w)$ computed by the run is correct.

[10] We use the *strong until* which requires that q eventually happens. If the weak *until* is used, the property is satisfied from both positions.

3

Timed Systems: Preliminaries

The passage from discrete to timed level of abstraction requires significant adaptations of the semantic domain, the logic and the automata. The first change involves considering behaviors that evolve on a time axis defined as a set of non-negative reals. The interaction between discrete events and dense time may give rise to certain well-known anomalies, such as “Zeno” behaviors, that should be carefully avoided. Consequently, in section 3.1, we define dense time Boolean signals as the semantic domain for timed systems. The behavioral correctness of timed systems does not rely only on the correctness of the output that they generate, but also on the actual timings where some discrete events are computed. Hence, we also need to extend the temporal logic LTL to enable expression of timing relations between subsequent events in the signal. In section 3.2 we introduce the real-time temporal logic MITL [AFH96] as our choice for specifying such properties. Finally, timed systems are usually modelled using timed automata [AD94], and in section 3.3 we describe *timed signal transducers* as an input/output variant of timed automata that will be used as temporal testers for MITL operators.

3.1 Signals

A signal over a domain D is a function $w : \mathbb{T} \to D$ where $\mathbb{T}$ is the time domain, which is either the set $\mathbb{R}_{\geq 0}$ of non-negative real numbers in the case of infinite signals or an interval $[0, r)$ if the signal is of finite length. We focus on the case where D is a finite domain, typically the set $\mathbb{B}^n$ of Boolean vectors over n variables. Each finite signal can be further decomposed into a *punctual signal*, defined only at 0 and denoted by $\dot{w}$, and an *open signal segment* defined over the interval $(0, r)$. We will denote such signal segments as $(w)_r$. The concatenation of a punctual signal and an open signal segment is a finite signal, and is simply their union. Concatenation of two finite signals w_1 and w_2 defined over $[0, r_1)$ and $[0, r_2)$, respectively, is the finite signal $w = w_1 \cdot w_2$, defined over $[0, r_1 + r_2)$ as

$$w[t] = \begin{cases} w_1[t] & \text{if } t < r_1 \\ w_2[t - r_1] & \text{otherwise} \end{cases}$$

A point-segment partition of $\mathbb{T}$ is an alternating sequence of adjacent points and open intervals of the form

$$J = \{t_0\}, (t_0, t_1), \{t_1\}, (t_1, t_2), \{t_2\}, \dots$$

with $t_0 = 0$ and $t_i < t_{i+1}$. With respect to such a given time partition, a signal w can be written as an alternating concatenation of points and open segments:

$$w = \dot{w}_0 \cdot (w_0)_{r_0} \cdot \dot{w}_1 \cdot (w_1)_{r_1} \cdots$$

where $\dot{w}_i$ is the value of the signal at t_i and $(w_i)_{r_i}$ is the segment which corresponds to the restriction of w to the interval (t_i, t_{i+1}) whose duration is $r_i = t_{i+1} - t_i$. An interval splitting is the act of partitioning a segment (t_i, t_{i+1}) into $(t_i, t'), \{t'\}, (t', t_{i+1})$. We say that a time partition J' is a refinement of J, denoted by $J' \prec J$ if it can be obtained from J by one or more interval splittings. A time partition is *compatible* with a signal w if the value of w is uniform in each open interval, that is, the segment $(w_i)_{r_i}$ is constant for every i.

The left and right limit of a signal w at point t are defined, as

$$w[\rightarrow t] = \lim_{r \rightarrow t^+} w[r] \quad \text{and} \quad w[t \leftarrow] = \lim_{r \rightarrow t^-} w[r],$$

respectively. We say that a time point t is *left-singular* with respect to w if $w[\rightarrow t] \neq w[t]$ and that it is *right-singular* if $w[t] \neq w[t\leftarrow]$. A point is singular if it is either left- or right-singular (or both). A point which is not singular is called *stationary*. Let us denote the sequence of singular points in w by $\mathcal{J}(w)$. A signal is *well-behaving* if the sequence $\mathcal{J}(w) = t_0, t_1, \dots$ is either finite or countable and diverging. In other words we exclude Zeno signals, those that change their value infinitely many times in a bounded time interval.

Every well-behaving signal w with $\mathcal{J}(w) = t_0, t_1, \dots$ induces a canonical time partition

$$J_w = \{t_0\}, (t_0, t_1), \{t_1\}, (t_1, t_2), \{t_2\}, \dots,$$

which is the coarsest time partition compatible with w (see Figure 3.1 for an example). In this case we can write the signal as

$$w = \dot{\sigma}_0 \cdot \sigma_0^{r_0} \cdot \dot{\sigma}_1 \cdot \sigma_1^{r_1} \cdot \dot{\sigma}_2 \cdot \sigma_2^{r_2} \cdots$$

where $\dot{\sigma}_i$ is the value at the singular point t_i and σ_i is the value of the signal in the interval (t_i, t_{i+1}). We will also use the notation

$$\sigma_0^{r_0} \cdot \sigma_1^{r_1} \cdot \sigma_2^{r_2} \cdots$$

when we do not care about the value at the singular points, that is, to denote the countably-many signals that agree on the open segments.

Signals can be combined and separated using the standard pairing and projection operators. Let $w_p : \mathbb{T} \rightarrow \mathbb{B}$, $w_q : \mathbb{T} \rightarrow \mathbb{B}$ and $w_{pq} : \mathbb{T} \rightarrow \mathbb{B}^2$ be signals. The pairing function is defined as

$$w_p \mid\mid w_q = w_{pq} \text{ if } \forall t \in \mathbb{T} \;\; w_{pq}[t] = (w_p[t], w_q[t])$$

and its inverse operation, projection as:

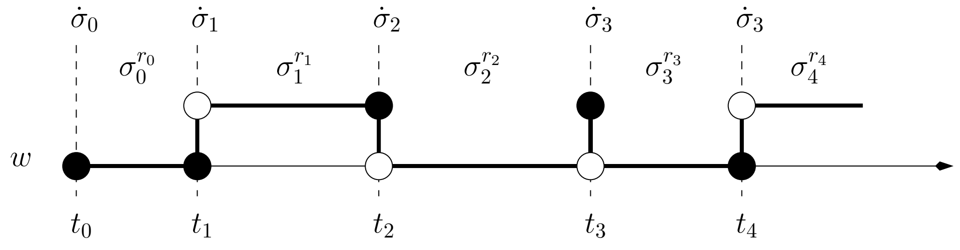


Fig. 3.1. The coarsest partition of a well-behaving signal w

$$w_p = w_{pq}|_p \quad w_q = w_{pq}|_q$$

Signal transductions are functions that map signals to signals. They can be memoryless such as the pointwise extensions of Boolean operations or more general ones realized by (timed) automata. The definition of *causal* signal transducers is similar to the definition for sequence transducers (Definition 2.1).

Note that the number of singular points in w_{pq} is at most the *sum* of the number of singular points in w_p and w_q, and that the number of singular points in $\textsc{op}(w_p, w_q)$, for a pointwise extension of a Boolean operator $\textsc{op}$ is at most that of w_{pq}. Hence well-behaving signals are closed under pairing, projection and Boolean operations.

The Minkowski sum $A \oplus B$ of two sets is the set $\{a + b : a \in A, b \in B\}$. In the special case of intervals one has $[a, b] \oplus [c, d] = [a + b, c + d]$. For intervals that may be open/close in either one of their sides, one can see that since $x < a$ and $y \leq b$ imply $x + y < a + b$, the Minkowski sum behaves on such intervals according to the following tables

$\oplus$	$[c$	$(c$
$[a$	$[a+c$	$(a+c$
$(a$	$(a+c$	$(a+c$

$\oplus$	$d)$	$d]$
$b)$	$b+d)$	$b+d)$
$b]$	$b+d)$	$b+d]$

Table 3.1. Minkowski sum

We use the notation $[a, b] \ominus [c, d] = [a - c, b - d] \cap \mathbb{T}$ to denote the Minkowski difference with saturation at zero and $t \oplus [a, b]$ as a shorthand for $\{t\} \oplus [a, b]$.

When considering signals of finite length $|w| = r$, we use the notation $w[t] = \bot$ when $t \geq |w|$. The restriction of a signal of length d is defined as

$$w' = \langle w \rangle_d \text{ iff } w'[t] = \begin{cases} w[t] \text{ if } t < d \\ \bot \quad \text{otherwise} \end{cases}$$

When we apply operations on signals of different lengths, we use the convention

$$\textsc{op}(v, \bot) = \textsc{op}(\bot, v) = \bot$$

which guarantees that if $w = \textsc{op}(w_1, w_2)$ then $|w| = \min(|w_1|, |w_2|)$.

The d*-suffix* of a signal w is the signal $w' = d \backslash w$ obtained from w by removing the prefix $\langle w \rangle_d$ from w, that is,

$$w'[t] = w[t + d] \text{ for every } t \in [0, |w| - d).$$

3.2 MITL: a Real-time Temporal Logic

The temporal logic MITL (metric interval temporal logic) is one of the most popular real-time extensions of LTL. It was originally introduced in [AFH96] as a restriction of the logic MTL [Koy90]. The principal modality of MITL is the *timed until* $\mathcal{U}_I$ where I is some non-punctual interval with integer or rational endpoints. A formula $p\mathcal{U}_{[a,b]}p$ is satisfied by a signal at any time instant t that admits q at some $t' \in [t+a, t+b]$, and where p holds continuously from t to t'. The original version of MITL contained only future temporal operators, although an investigation of past and future versions of MITL was carried out in [AH92b].

3.2.1 Syntax, Semantics and Rewriting Rules

We consider the MITL logic with both *future* and *past* operators. The syntax of MITL is defined by the grammar

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1\, \mathcal{U}_I\, \varphi_2 \mid \varphi_1\, \mathcal{S}_I\, \varphi_2$$

where p belongs to a set $P = \{p_1, \dots, p_n\}$ of propositions and I is an interval of the form $[a,b],\ [a,b),\ (a,b],\ (a,b),\ [a,\infty)$ or (a,∞) where $0 \le a < b$ are rational numbers.[1] As in LTL the basic MITL operators can be used to derive other standard Boolean and temporal operators, in particular the time-constrained *eventually*, *once*, *always*, and *historically* operators:

$$\begin{array}{ll} \Diamond_I\, \varphi = \top\ \mathcal{U}_I\ \varphi & \diamondminus_I\, \varphi = \top\ \mathcal{S}_I\ \varphi \\ \Box_I\, \varphi = \neg\Diamond_I\, \neg\varphi & \boxminus_I\, \varphi = \neg\diamondminus_I\, \neg\varphi \end{array}$$

The semantics of an MITL formula φ with respect to an n-dimensional Boolean signal w is described via the satisfiability relation $(w,t) \models \varphi$, indicating that the signal w satisfies φ at time t, according to following recursive definition.

$$\begin{array}{lcl} (w,t) \models p & \leftrightarrow & p[t] = 1 \\ (w,t) \models \neg\varphi & \leftrightarrow & (\xi,t) \not\models \varphi \\ (w,t) \models \varphi_1 \vee \varphi_2 & \leftrightarrow & (w,t) \models \varphi_1 \text{ or } (w,t) \models \varphi_2 \\ (w,t) \models \varphi_1\ \mathcal{U}_I\ \varphi_2 & \leftrightarrow & \exists\, t' \in t \oplus I\ (\xi,t') \models \varphi_2 \text{ and} \\ & & \forall\, t'' \in (t,t')\ (w,t'') \models \varphi_1 \\ (w,t) \models \varphi_1\ \mathcal{S}_I\ \varphi_2 & \leftrightarrow & \exists\, t' \in t \ominus I\ (w,t') \models \varphi_2 \text{ and} \\ & & \forall\, t'' \in (t',t)\ (w,t'') \models \varphi_1 \end{array} \tag{3.1}$$

A formula φ is satisfied by w if $(w,0) \models \varphi$. Recall that the satisfaction relation can be viewed as characteristic function χ^φ mapping signals over $\mathbb{B}^n$ into Boolean signals, such that $u = \chi^\varphi(w)$ meaning that for every $t \ge 0$, $u[t] = 1$ if and only if $(w,t) \models \varphi$. The definitions of $\mathcal{U}_I$ and $\mathcal{S}_I$ are *strict* as originally proposed in [AFH96], meaning that φ_1 need not hold at t and neither at the moment t' when φ_2 becomes true. Note also that when I is left-open with a bound a, the truth of φ_2 at $t+a$ does not count for satisfaction.

[1] As a general remark concerning timed automata and logics, by proper calibration of the time step, every finite MITL formula and any finite timed automaton can be converted to such where a and b are integers.

Let us remark that the original logic MTL [Koy90] for which MITL is restriction allows also “punctual” intervals of the form $[a, a]$ in the temporal modalities. To see why this is problematic in dense time consider the operator $\diamondminus_{[a,a]}$ that we denote from now on as $\diamondminus_a$. This operator, viewed as a signal transducer is a *shift*: its output at t is the value of its input at time $t - a$. To realize this operator we need a device which can “memorize” the value of the input signal in a time window of length a. Without further assumptions on the signal, such a memorization is beyond the capabilities of any automaton with a finite number of states and clocks. The same applies to the future operator $\diamond_a$ with the additional complication of handling predictions. The good news, however, is that if one knows in advance a bound on the *variability* of the input, this operator can be realized by a finite timed automaton. We will make use of this fact in the sequel.

Untimed *strict* temporal operators $\mathcal{U}$ and $\mathcal{S}$ can be expressed using the timed operators where the interval is $(0, \infty)$ Similarly, we can define *non-strict* untimed temporal operators $\underline{\mathcal{U}}$ and $\underline{\mathcal{S}}$ (which are the commonly-used interpretations of $\mathcal{U}$ and $\mathcal{S}$ in LTL) in terms of the strict ones.

$$\begin{array}{ll}
\varphi_1\ \mathcal{U}\ \varphi_2 = \varphi_1\ \mathcal{U}_{(0,\infty)}\ \varphi_2 & \varphi_1\ \mathcal{S}\ \varphi_2 = \varphi_1\ \mathcal{S}_{(0,\infty)}\ \varphi_2 \\
\varphi_1\ \underline{\mathcal{U}}\ \varphi_2 = \varphi_2 \vee (\varphi_1 \wedge (\varphi_1\ \mathcal{U}\ \varphi_2)) & \varphi_1\ \underline{\mathcal{S}}\ \varphi_2 = \varphi_2 \vee (\varphi_1 \wedge (\varphi_1\ \mathcal{S}\ \varphi_2))
\end{array}$$

Note that $\underline{\mathcal{U}}$ differs from $\mathcal{U}_{[0,\infty)}$.

In what follows we show that some of the timed operators ($\mathcal{U}_I$ and $\mathcal{S}_I$, each with all types of intervals) can be written in terms of simpler ones, which will allow us to simplify our monitoring and verification procedures for MITL. We start with the following lemma, proved also in [DT04, MNP06], which shows that the timed *until* can be expressed by a combination of untimed *until* and timed *eventually*.

Lemma 3.1 ($\mathcal{U}_I$ can be expressed by $\mathcal{U}$ and $\diamond_I$). *For every signal* w,

$$\begin{array}{lcl}
w \models \varphi_1\ \mathcal{U}_{(a,b)}\ \varphi_2 & \leftrightarrow & w \models \square_{(0,a]}\,\varphi_1 \wedge \square_{(0,a]}(\varphi_1\ \mathcal{U}\ \varphi_2) \wedge \diamond_{(a,b)}\,\varphi_2 \\
w \models \varphi_1\ \mathcal{U}_{(a,b]}\ \varphi_2 & \leftrightarrow & w \models \square_{(0,a]}\,\varphi_1 \wedge \square_{(0,a]}(\varphi_1\ \mathcal{U}\ \varphi_2) \wedge \diamond_{(a,b]}\,\varphi_2 \\
w \models \varphi_1\ \mathcal{U}_{[a,b)}\ \varphi_2 & \leftrightarrow & w \models \square_{(0,a)}\,\varphi_1 \wedge \square_{(0,a]}(\varphi_1\ \underline{\mathcal{U}}\ \varphi_2) \wedge \diamond_{[a,b)}\,\varphi_2 \\
w \models \varphi_1\ \mathcal{U}_{[a,b]}\ \varphi_2 & \leftrightarrow & w \models \square_{(0,a)}\,\varphi_1 \wedge \square_{(0,a]}(\varphi_1\ \underline{\mathcal{U}}\ \varphi_2) \wedge \diamond_{[a,b]}\,\varphi_2 \\
w \models \varphi_1\ \mathcal{U}_{(a,\infty)}\ \varphi_2 & \leftrightarrow & w \models \square_{(0,a]}\,\varphi_1 \wedge \square_{(0,a]}(\varphi_1\ \mathcal{U}\ \varphi_2) \\
w \models \varphi_1\ \mathcal{U}_{[a,\infty)}\ \varphi_2 & \leftrightarrow & w \models \square_{(0,a)}\,\varphi_1 \wedge \square_{(0,a]}(\varphi_1\ \underline{\mathcal{U}}\ \varphi_2)
\end{array}$$

$$\begin{array}{lcl}
w \models \varphi_1\ \mathcal{S}_{(a,b)}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a]}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \mathcal{S}\ \varphi_2) \wedge \diamondminus_{(a,b)}\,\varphi_2 \\
w \models \varphi_1\ \mathcal{S}_{(a,b]}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a]}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \mathcal{S}\ \varphi_2) \wedge \diamondminus_{(a,b]}\,\varphi_2 \\
w \models \varphi_1\ \mathcal{S}_{[a,b)}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a)}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \underline{\mathcal{S}}\ \varphi_2) \wedge \diamondminus_{[a,b)}\,\varphi_2 \\
w \models \varphi_1\ \mathcal{S}_{[a,b]}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a)}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \underline{\mathcal{S}}\ \varphi_2) \wedge \diamondminus_{[a,b]}\,\varphi_2 \\
w \models \varphi_1\ \mathcal{S}_{(a,\infty)}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a]}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \mathcal{S}\ \varphi_2) \\
w \models \varphi_1\ \mathcal{S}_{[a,\infty)}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a)}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \underline{\mathcal{S}}\ \varphi_2)
\end{array}$$

Proof. We prove the first of these identities, the others are similar. One direction of the equivalence follows directly from the semantics of timed *until*, so we consider only the other direction which is proved via the following observations:

1. If $w \models \Box_{(0,a]} \varphi_1$, then φ_1 holds continuously throughout $(0, a]$
2. If $w \models \Box_{(0,a]}(\varphi_1 \ \mathcal{U} \ \varphi_2)$, then $\varphi_1 \ \mathcal{U} \ \varphi_2$ has to hold at a and there exists $t' > a$ such that φ_2 is *true* and φ_1 holds during (a, t')
3. Formula $\Diamond_{(a,b)} \varphi_2$ requires the existence of $t' \in (a, b)$ such that φ_2 holds at t'

Combining these observations we can see that $w \models \Box_{(0,a]} \varphi_1 \wedge \Box_{(0,a]}(\varphi_1 \ \mathcal{U} \ \varphi_2) \wedge \Diamond_{(a,b)} \varphi_2$ implies that there exists $t' \in (a, b)$ such that φ_2 is *true* at t' and φ_1 holds continuously during $(0, t')$, which is exactly the semantic definition of $\varphi_1 \ \mathcal{U}_{(a,b)} \ \varphi_2$. $\Box$

Consequently, the operators $\mathcal{U}$, $\mathcal{S}$, $\Diamond_I$ and $\diamondminus_I$, where I ranges over the interval types $[a, b]$, $[a, b)$, $(a, b]$ and (a, b), are sufficient to express any MITL property. However we can still reduce the number and complexity of tester types using the following observation, first made in [AFH96], which says that the time instants in which a property of the form $\Diamond_{[a,b]} p$ is satisfied are unions of intervals, each with duration not smaller than $b - a$.

Lemma 3.2 (Bounded Variability[2]). *Let $u = \chi^{\Diamond_{[a,b]} p}(w)$ for an arbitrary signal w. Then in any decomposition of u having the form $u = u' \cdot 0^{r'} \cdot \dot{1} \cdot 1^r \cdot u''$ or $u = u' \cdot 0^{r'} \cdot \dot{0} \cdot 1^r \cdot u''$, we have $r \geq b - a$.*

Proof. We prove for the first case. Let t be the duration of the prefix $u' \cdot 0^{r'}$, and hence t is the *earliest* point in its neighborhood where $\Diamond_{[a,b]} p$ holds. This means that p holds at $t + b$ and hence $\Diamond_{[a,b]} p$ will hold throughout the interval $t \oplus [0, b - a]$. $\Box$

Consequently, we can use identities of the form $\Diamond_{[a,b]} = \Diamond_a \Diamond_{[0,b-a]}$ to decompose $\Diamond_{[a,b]}$ into $\Diamond_a$, and $\Diamond_I$ with $I \in \{[0, a], (0, a], [0, a), (0, a)\}$ where $\Diamond_a$ will be applied only to signals of bounded variability satisfying Lemma 3.2. Moreover, observing that

$$\Diamond_{[0,a]} \varphi \quad = \quad \varphi \vee \Diamond_a (\diamondminus_{(0,a)} \varphi \vee \varphi)$$

we can conclude the following proposition which will be used in chapter 8:

Proposition 3.3 (Basic MITL Operators). *Any MITL formula can be rewritten into a form which uses only the temporal operators $\mathcal{U}$, $\mathcal{S}$, $\Diamond_{(0,a)}$, $\diamondminus_{(0,a)}$, $\Diamond_a$ and $\diamondminus_a$, with the last two applied to sub-formulae whose characteristic functions are of uniform bounded variability.*

Expressing Events

MITL does not provide constructs that allow to reason explicitly about *instantaneous events* which can be viewed as taking place in singular intervals of zero duration. A natural way to introduce them is to consider the instants when a signal changes its value. To this end we propose two unary operators, *rise* $\uparrow$ and *fall* $\downarrow$, which hold at the rising and falling edges of a Boolean signal, respectively. However, since we allow singular points to be equal to their left neighborhood, $\uparrow p$ may hold at t even if $p[t] = 0$ as illustrated in Figure 3.2. Intuitively, $\uparrow \varphi$ holds at t if φ is false at t and true in a right neighborhood

[2] A similar claim holds for the corresponding past operator.

of t, or if φ is true at t and false in a left neighborhood of t. These operators can be expressed in MITL if we allow *both* future and past operators, as follows:

$$\uparrow\varphi = (\varphi \wedge (\neg\varphi\,\mathcal{S}\,\mathrm{T}))\ \vee\ (\neg\varphi \wedge (\varphi\,\mathcal{U}\,\mathrm{T}))$$
$$\downarrow\varphi = (\neg\varphi \wedge (\varphi\,\mathcal{S}\,\mathrm{T}))\ \wedge\ (\varphi \wedge (\neg\varphi\,\mathcal{U}\,\mathrm{T}))$$

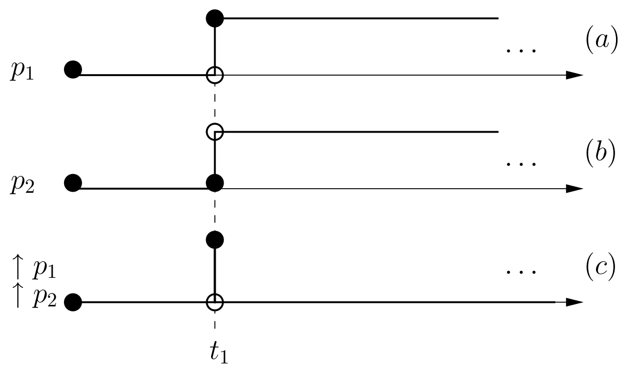


Fig. 3.2. Two signals p_1 and p_2 that differ at time t where both $\uparrow p_1$ and $\uparrow p_2$ hold.

3.2.2 Interpretation of MITL over Incomplete Behaviors

The problems related to finitary interpretation of LTL discussed in section 2.2 are inherited by dense-time adaptations of temporal logic. We adopt the approach of quantifying over time within the length of the finite behavior w and accordingly adapt the semantics of the until $\mathcal{U}_I$ operator as follows:

$$\begin{array}{lcl}(w,t) \models \varphi_1\ \mathcal{U}_I\ \varphi_2 & \leftrightarrow & \exists\, t' \in t \oplus I \text{ st } (t' < |w| \text{ and } (\xi, t') \models \varphi_2) \text{ and} \\ & & \forall\, t'' \in (t, t')\ (w, t'') \models \varphi_1\end{array}$$

Intuitively, this definition[3] gives a *strong* interpretation of *until* which requires that φ_2 will eventually hold within the interval I and before the end of the trace. This definition allows to derive other standard future timed operators *eventually* and *always* in the usual fashion

$$\Diamond_I\,\varphi = \mathrm{T}\ \mathcal{U}_I\ \varphi \qquad \Box_I\,\varphi = \neg\Diamond_I\,\neg\varphi$$

where $\Diamond_I\,\varphi$ remains a *strong* operator (eventuality has to hold within I and before the end of the behavior), while $\Box_I\,\varphi$ becomes a *weak* operator requiring that φ holds throughout I within the length of the trace.

It is not hard to see that this finitary definition of the timed until operator preserves the same simplification rules presented in Lemma 3.1. Finally, we can note that the PSL

[3] If w is an infinite behavior, this definition of $\varphi_1\mathcal{U}_I\varphi_2$ is equivalent to the one of section 3.2

approach of providing an alternative *weak* ($\mathcal{U}_I^w$) version of timed until requires minimal effort in adapting the operator semantics:

$$(w,t) \models \varphi_1 \; \mathcal{U}_I^w \; \varphi_2 \quad \leftrightarrow \quad \begin{array}{l} \exists\, t' \in t \oplus I \text{ st } (t' \geq |w| \text{ or } (\xi, t') \models \varphi_2) \text{ and} \\ \forall\, t'' \in (t,t')\; (w,t'') \models \varphi_1 \end{array}$$

3.2.3 Some Properties of $p\mathcal{S}q$ and $p\mathcal{U}q$

In this section we prove some semantic properties of $p\mathcal{S}q$ and $p\mathcal{U}q$. In particular, we show that their satisfiability is uniform in all open time segments where their input does not change.

Lemma 3.4 (Since is Left-continuous). *Let* $u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdots = \chi^{p\mathcal{S}q}(w)$. *Then,* $\dot{u}_0 = 0$ *and for any* $i \geq 1$, $\dot{u}_i = u_{i-1}$.

Proof. The proof for $\dot{u}_0 = 0$ is trivial and follows directly the semantics of $p\mathcal{S}q$ evaluated at time 0, whose satisfaction requires the existence of $t' < 0$ which is not the case. For $i \geq 1$, assume first that $\dot{u}_i = 1$. Then there exist $t' < t_i$ such that q is satisfied at t' and that p holds continuously throughout the interval (t', t_i). Then, it follows that $(w,t) \models p\mathcal{S}q$ everywhere in (t', t_i) and, consequently $u_{i-1} = 1 = \dot{u}_i$. If $\dot{u}_i = 0$, there are two possibilities, either q was never true at any $t' \in [0, t_i)$, and hence u was false in the whole interval $(0, t_i)$, or that for any $t'' \in [0, t_i)$ where q was true, there is $t' \in (t'', t_i)$ where p was false, implying that $p\mathcal{S}q$ was not satisfied at (t', t_i) and $u_{i-1} = 0 = \dot{u}_i$. □

Lemma 3.5 (Until is Right-continuous). *Let* $u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdot = \chi^{p\mathcal{U}q}(w)$. *Then, for any* $i \geq 0$, $\dot{u}_i = u_i$.

Proof. Assume first that $\dot{u}_i = 1$. Then there exists $t' > t_i$ such that q is satisfied at t' and that p holds continuously throughout the interval (t_i, t'). Then, it follows that $(w,t) \models p\mathcal{U}q$ everywhere in (t_i, t') and, consequently $u_i = 1 = \dot{u}_i$. If $\dot{u}_i = 0$, there are two possibilities, either q never becomes true at any $t' > t_i$ and hence u is false in the whole open interval (t, ∞), or for any $t'' > t_i$ where q is true there is $t' \in (t_i, t'')$ where p does not hold which implies that $p\mathcal{U}q$ is not satisfied at (t_i, t') and $u_i = 0 = \dot{u}_i$. □

Lemma 3.6 (Semantic Rules for Since). *Let* $w = \dot{w}_0 \cdot (w_0)_{r_0} \cdot \dot{w}_1 \cdot (w_1)_{r_1} \cdots$ *be a finite or infinite signal and let* $u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdots = \chi^{p\mathcal{S}q}(w)$. *Then, for every* $i \geq 0$,

1. *if* $w_i = \overline{p}$, *then* $u_i = 0$,
2. *if* $w_i = pq$, *then* $u_i = 1$
3. *if* $w_i = p\overline{q}$, *there are three possibilities:*
   a) *if* $\dot{w}_i = \overline{pq}$, *then* $u_i = 0$
   b) *if* $\dot{w}_i = q$, *then* $u_i = 1$
   c) *if* $\dot{w}_i = p\overline{q}$, *then* $u_i = \dot{u}_i$.

Case	$\dot{w}_i$	w_i	u_i
1	$*$	$\overline{p}$	0
2	$*$	pq	1
3a	$\overline{pq}$		0
3b	q	$p\overline{q}$	1
3c	$p\overline{q}$		$\dot{u}_i$

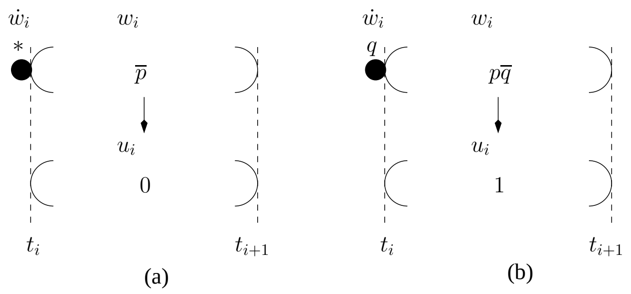


Fig. 3.3. $p\,\mathcal{S}\,q$ rules for determining u_i and examples when (a) $w_i = \overline{p}$ and (b) $w_i = p\overline{q}$ and $\dot{w}_i = q$

Proof. The value of u in the i^{th} segment is determined with respect to the values of inputs p and q in the same segment w_i and at the preceding singular point $\dot{w}_i$. It is not hard to see that the 5 cases for values of $\dot{w}_i$ and w_i shown in Figure 3.3 cover all 16 possible combinations of values for p and q at the i^{th} singular point and the adjacent open segment. For any $t \in (t_i, t_{i+1})$ in the i^{th} segment, we have

Case 1: For any $t' < t$ which is in (t_i, t_{i+1}), by definition p does not hold throughout (t', t), hence $(w, t) \not\models p\mathcal{S}q$, that is $u_i = 0$.
Case 2: There exists $t' < t$ which is also in (t_i, t_{i+1}), where by definition q holds at t' and p holds continuously throughout (t', t). Hence $(w, t) \models p\mathcal{S}q$ for all such t and $u_i = 1$.
Case 3-(a): p was false at t_i and q does not hold anywhere in the interval (t_i, t), which implies that $p\mathcal{S}q$ is not satisfied throughout (t_i, t_{i+1}) and $u_i = 0$.
Case 3-(b): q was true at t_i and p was continuously true during (t_i, t), implying that $p\mathcal{S}q$ is satisfied at (t_i, t_{i+1}) and $u_i = 1$.
Case 3-(c): p holds and q remains false throughout $[t_i, t)$. Hence, $p\mathcal{S}q$ holds at t iff there is $t' \in [0, t_i)$ where q holds, and p remains true during (t', t_i), that is iff $p\mathcal{S}q$ holds at t_i. This implies that $p\mathcal{S}q$ is satisfied at (t_i, t_{i+1}) iff it is satisfied at t_i and $u_i = \dot{u}_i$.
□

Lemma 3.7 (Semantic Rules for Until). *Let* $w = \dot{w}_0 \cdot (w_0)_{r_0} \cdot \dot{w}_1 \cdot (w_1)_{r_1} \cdots$ *be a finite or infinite signal and let* $u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdots = \chi^{p\mathcal{U}q}(w)$*. Then, for every* $i \geq 0$*,*

1. *if* $w_i = \overline{p}$*, then* $u_i = 0$*,*
2. *if* $w_i = pq$*, then* $u_i = 1$
3. *if* $w_i = p\overline{q}$*, then either* w_i *is the last segment in* w *and* $u_i = 0$*, or:*
   a) *if* $\dot{w}_{i+1} = \overline{pq}$*, then* $u_i = 0$
   b) *if* $\dot{w}_{i+1} = q$*, then* $u_i = 1$
   c) *if* $\dot{w}_{i+1} = p\overline{q}$*, then* $u_i = \dot{u}_i$*.*

Proof. The value of u in the i^{th} segment is determined with respect to the values of inputs p and q in that same segment w_i and the next singular point $\dot{w}_{i+1}$. It is not hard to see that the 5 cases for values of w_i and $\dot{w}_{i+1}$ cover all 16 possible combinations of values for p and q at w_i and $\dot{w}_{i+1}$. For any $t \in (t_i, t_{i+1})$ in the i^{th} segment, we have

Case	u_i	w_i	$\dot{w}_{i+1}$
1	0	$\overline{p}$	$*$
2	1	pq	$*$
3a	0		$\overline{pq}$
3b	1	$p\overline{q}$	q
3c	$\dot{u}_{i+1}$		$p\overline{q}$

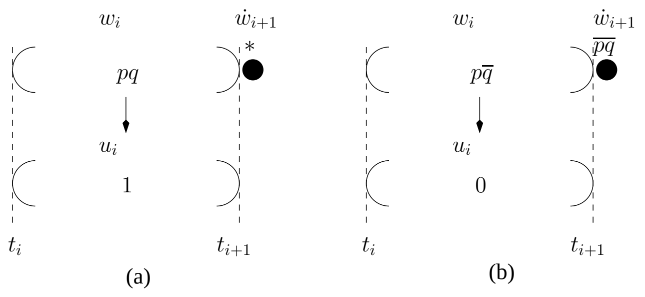


Fig. 3.4. $p\mathcal{U}q$ rules for determining u_i and examples when (a) $w_i = pq$ and (b) $w_i = p\overline{q}$ and $\dot{w}_{i+1} = \overline{pq}$

Case 1: For any $t' > t$ in (t_i, t_{i+1}), and by definition p does not hold throughout (t, t'), hence $(w, t) \not\models p\mathcal{U}q$ and $u_i = 0$.

Case 2: There exists $t' > t$ in (t_i, t_{i+1}) such that by definition q holds at t' and p holds continuously throughout (t, t'). Hence $(w, t) \models p\mathcal{U}q$ for all such t and $u_i = 1$.

Case 3-(a): By definition p is false at t_{i+1} and q does not hold anywhere in the interval (t, t_{i+1}), which implies that $p\mathcal{U}q$ is not satisfied throughout (t_i, t_{i+1}) and $u_i = 0$.

Case 3-(b): q is true at t_{i+1} and p continuously holds during (t, t_{i+1}), implying that $p\mathcal{U}q$ is satisfied at (t_i, t_{i+1}) and $u_i = 1$.

Case 3-(c): p holds and q remains false throughout $(t, t_{i+1}]$. Hence, $p\mathcal{U}q$ holds at t iff there is $t' > t_{i+1}$ where q holds, and p remains true during (t_{i+1}, t'), that is iff $p\mathcal{U}q$ holds at t_{i+1}. This implies that $p\mathcal{U}q$ is satisfied at (t_i, t_{i+1}) iff it is satisfied at t_{i+1} and $u_i = \dot{u}_{i+1}$.

The only remaining case is when $w_i = p\overline{q}$ and it is the last segment in the signal w (end of the signal if w is of finite length or w_i is defined over (t_i, ∞) if w is infinite). Since in both cases there is no $t' > t_i$ where q is true, $u_i = 0$. □

3.3 Timed Automata

Timed systems are usually modeled with *timed automata* [AD94], which are automata augmented with auxiliary clock variables. Clocks may be reset to zero upon certain transitions, and while the automaton remains in a state, their values advance uniformly, thus indicating the time elapsed since their respective resetting events. Clock conditions may appear as transition "guards" thus restricting transitions to take place when these conditions are met. This way, timed automata refine ordinary automata by letting them be sensitive not only to the logical form of their input signals but also to their metric aspects, that is, the distance between events.

In this thesis, timed automata are used for the building temporal testers for basic MITL operators described in Proposition 3.3. These basic testers can then be composed in order to build testers for arbitrary MITL formula, and this translation will be presented in chapter 8. We start by describing the way clock variables and the input/output alphabet can be referred to in timed automata. Our definition deviate slightly from older definitions of timed automata [AD94, HNSY94, Alu99] as well as from our own [MNP06] mainly because we consider them as transducers where input as well as output symbols are associated with both states *and* transitions. This allows us to synchronize in a clean way the runs of the automaton (and their induced point-segment time partitions) with the input and output signals. We consider a set $\mathcal{C} = \{x_1, \ldots, x_n\}$ of clock variables each ranging over $\mathbb{R}_{\geq 0} \cup \{\bot\}$ where $\bot$ is a special symbol indicating that the clock is currently *inactive*.[4] and extend the order relation on $\mathbb{R}_{\geq 0}$ accordingly by letting $\bot < v$ for every $v \in \mathbb{R}_{\geq 0}$.

The set of *clock valuations*, each denoted as $v = (v_1, \ldots, v_n)$, defines the clock space $\mathcal{H} = (\mathbb{R}_{\geq 0} \cup \{\bot\})^n$. A *configuration* of a timed automaton is a pair of the form (q, v) with q being a discrete state. For a clock valuation $v = (v_1, \ldots, v_n)$, $v + t$ is the valuation $(v'_1, \ldots, v'_n)$ such that $v'_i = v_i$ if $v_i = \bot$ and $v'_i = v_i + t$ otherwise. It represents the values of the clocks after spending t time in a state starting from valuations v. A *clock constraint* is a Boolean combination of conditions of the forms $x < d$, $x \leq d$, $x \geq d$ or $x > d$ for some integer d.

Definition 3.8 (Timed Signal Transducers). *A timed signal transducer is a tuple* $\mathcal{A} = (\Sigma, \Gamma, Q, \mathcal{C}, I, \Delta, \lambda, \gamma, q_{in}, F)$ *where:*

1. *Σ is the input alphabet and Γ is the output alphabet*
2. *Q is a finite set of discrete states (locations)*
3. *$\mathcal{C}$ is a finite number of clocks*
4. *The staying condition (invariant) I assigns to every location q a subset I_q of $\mathcal{H}$ defined by a conjunction of inequalities of the form $x \leq d$ or $x < d$, for some clock x and integer d.*
5. *The transition relation Δ consists of elements of the form $\delta = (q, g, \rho, q')$ where*
   - *q and q' locations*
   - *the transition guard g is a subset of $\mathcal{H}$ defined by a clock constraint*

[4] This is syntactic sugar since clock inactivity in a state can be encoded implicitly by the fact that in all paths emanating from the state, the clock is reset to zero before being tested [DY96].

- *ρ is the update function, a transformation of $\mathcal{H}$ defined by one or more assignments of the form $x := 0$ or $x := \perp$ for a clock x, as well as by* copy assignments *of the form $x_i := x_j$*

6. *The input labeling function $\lambda : Q - \{q_{in}\} \cup \Delta \rightarrow 2^{\Sigma}$ associates a subset of the input alphabet to every location and transition*
7. *The output labeling function $\gamma : Q - \{q_{in}\} \cup \Delta \rightarrow \Gamma$ assigns output letters to each location*[5] *and transition*
8. *$q_{in} \subseteq Q$ is the initial state*
9. *$\mathcal{F} \subseteq 2^{Q \cup \Delta}$ is a generalized Büchi acceptance condition on states and transitions.*

Intuitively a *run* of a timed automaton consists of an alternation of *discrete steps* where a transition whose guard is satisfied is taken, and *time steps* where the automaton stays in a state for some duration provided that I_q holds. For transducers we need to establish a relation between a run of the automaton, an input signal w which induces it and an output signal u which is produced during the run. First, we associate via λ a subset of the input alphabet to each location and transition. During a time step of duration r in a location q, the automaton reads an open segment $(w)_r$ of w in which the values are required to belong to $\lambda(q)$. While taking a transition δ, the automaton reads a point in the signal whose value should belong to $\lambda(\delta)$. Likewise, we associate an output symbol (either 0 or 1 in the case of our temporal testers) with each state and transition. The whole output signal is constructed by concatenating points and open segments collected during the run.

Formally, a *step* of the automaton is one of the following:

- **A time step:** $(q, v) \xrightarrow{(w)_r/\tau^r} (q, v + r)$, where $r \in \mathbb{R}_+$, all the letters appearing in the segment $(w)_r$ are in $\lambda(q)$, $\tau = \gamma(q)$, and for all $v' \in (v, v+r)$, v' satisfies the staying condition I_q;
- **A discrete step:** $(q, v) \xrightarrow{\dot{w}/\dot{\tau}} (q', v')$, for some transition $\delta = (q, g, \rho, q') \in \Delta$, such that $v \in g$, $v' = \rho(v)$, $w \in \lambda(\delta)$ and $\tau = \gamma(\delta)$.

A *run* of the automaton starting from the initial configuration $(q_{in}, \perp)$ and induced by an input signal w is a finite or infinite sequence of alternating time and discrete steps of the form

$$\xi : (q_{in}, \perp) \xrightarrow{\dot{w}_0/\dot{\tau}_0} (q_0, v_0) \xrightarrow{(w_0)_{r_0}/\tau_0^{r_0}} (q_0, v_0 + r_0) \xrightarrow{\dot{w}_1/\dot{\tau}_1} (q_1, v_1) \xrightarrow{(w_1)_{r_1}/\tau_1^{r_1}} (q_1, v_1 + r_1) \cdots$$

such that $\sum r_i$ diverges and

$$w = \dot{w}_0 \cdot (w_0)_{r_0} \cdot \dot{w}_1 \cdot (w_1)_{r_1} \cdots .$$

The output of the run is the signal

$$u = \dot{\tau}_0 \cdot \tau_0^{r_0} \cdot \dot{\tau}_1 \cdot \tau_1^{r_1} \cdots .$$

A run is accepting if for every $F \in \mathcal{F}$, the set of absolute time instants in which it visits states in F or makes transitions in F is unbounded. The automaton realizes a

[5] The initial location is excluded as it serves only as a source for the first transition.

sequential relation $R_{\mathcal{A}}$ on signals over its input and output alphabets defined by $(w, u) \in R_{\mathcal{A}}$ iff there is an accepting run induced by input signal w which produces the output signal u. In chapter 8 we build such transducers for the MITL operators and show that they constitute total functions from input to output which are exactly the characteristic functions of their respective operators.

Before presenting the testers for the temporal operators, we illustrate the way timed transducers work on the simplest example, a tester for the property p, depicted in Figure 3.5-(a). We use a statechart notation where some states may be grouped into macro-states (dashed lines) so that a transition outgoing from a macro-state represents several identical transitions outgoing from all the state in it and, likewise, a transition entering a macro-state represents several identical transitions that go into all its states. We omit the initial state q_{in} from all the figures and the transition from it (which must take place at time zero) appear as sourceless.

Consider the finite signal $w = \overline{p} \cdot \overline{p}^{r_0} \cdot p \cdot p^{r_1} \cdot p \cdot \overline{p}^{r_2}$ depicted at Figure 3.5-(b). Its run on the automaton, which does not use clocks as all since this is an instantaneous operator, can be written as

$$(q_{in}, \perp) \xrightarrow{\overline{p}/\overline{u}} (s_0, \perp) \xrightarrow{\overline{p}^{r_0}/\overline{u}^{r_0}} (s_0, \perp) \xrightarrow{p/u} (s_1, \perp) \xrightarrow{p^{r_1}/u^{r_1}} (s_1, \perp) \xrightarrow{p/u} (s_0, \perp) \xrightarrow{\overline{p}^{r_2}/\overline{u}^{r_2}} (s_0, \perp),$$

and illustrated in Figure 3.5-(c). The automaton exhibits "infinitesimal" non determinism: during any moment t in a time step, the automaton may initiate a transition to the other state. If the value of the signal in the adjacent open segment starting at t conforms with the value of that state, the run is continued from there, otherwise it is aborted immediately and the automaton continues with the time step.

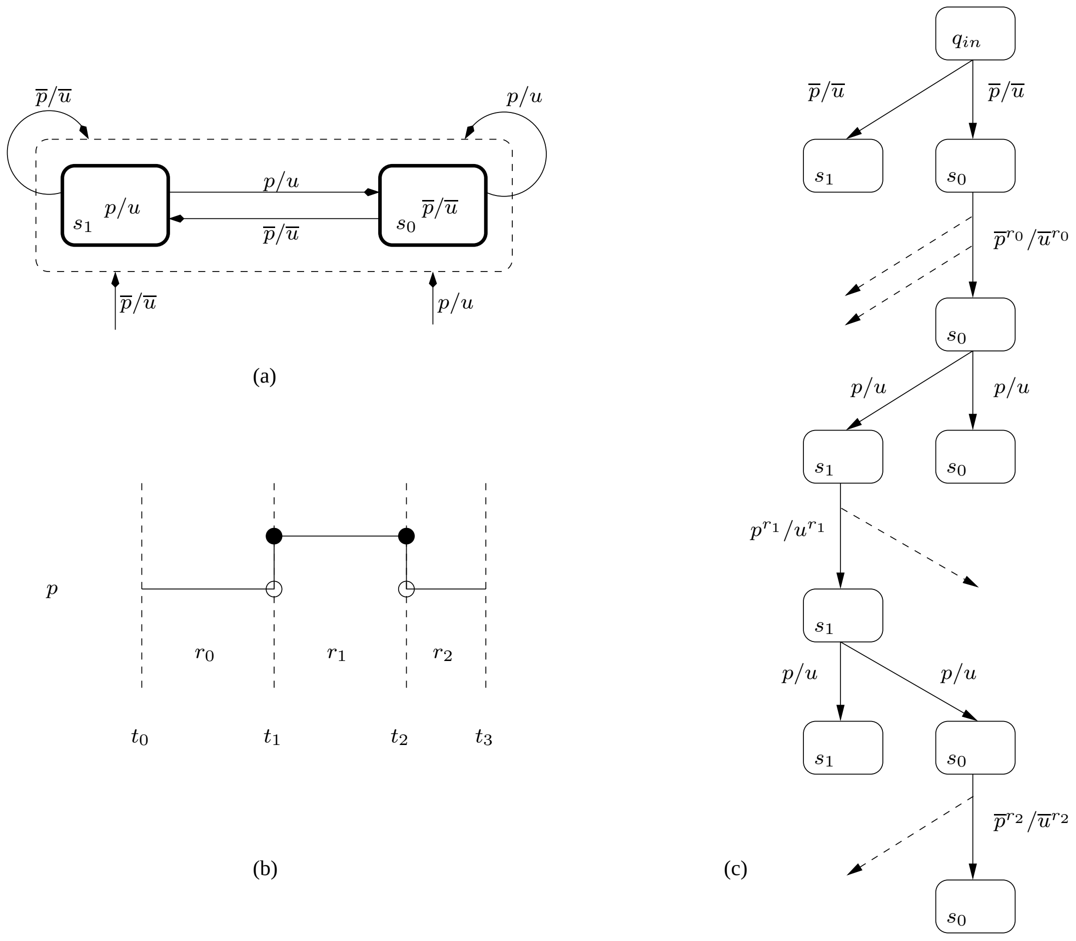

$\overline{p}/\overline{u}$
p/u
p/u
s_1
p/u
s_0
$\overline{p}/\overline{u}$
$\overline{p}/\overline{u}$
$\overline{p}/\overline{u}$
p/u
(a)
p
r_0
r_1
r_2
t_0
t_1
t_2
t_3
(b)
q_{in}
$\overline{p}/\overline{u}$
$\overline{p}/\overline{u}$
s_1
s_0
$\overline{p}^{r_0}/\overline{u}^{r_0}$
s_0
p/u
p/u
s_1
s_0
p^{r_1}/u^{r_1}
s_1
p/u
p/u
s_1
s_0
$\overline{p}^{r_2}/\overline{u}^{r_2}$
s_0
(c)

Fig. 3.5. (a) The temporal tester for p; (b) A signal $w = \overline{p} \cdot \overline{p}^{r_0} \cdot p \cdot p^{r_1} \cdot p \cdot \overline{p}^{r_2}$; (c) The run of the automaton on w. Some of the aborted runs are shown explicitly and some are illustrated by the dashed lines.

4

Monitoring Timed Behaviors

In this section, we describe two procedures for monitoring timed MITL properties. These procedures are:

1. An *offline* procedure that propagates truth values upwards from propositions via super-formulae up to the main formula. The offline monitoring method is presented in section 4.1
2. An *incremental* marking procedure that updates the marking each time a new segment of the input signal is observed. Section 4.2 describes the incremental monitoring algorithm.

Unlike automata-based monitoring algorithms, the procedures that we propose are directly applied to signals. A central notion in these algorithms is that of the *satisfaction signal* $u_\varphi = \chi^\varphi(w)$ associated with a formula φ and a signal w. We remind the reader that this signal satisfies $u_\varphi[t] = 1$ iff $(w, t) \models \varphi$. Due to the non-causality of future operators of MITL, the value of $u_\varphi[t]$ is not necessarily known at time t, that is, after observing $w[t]$, and may depend on future values of w.

4.1 Offline Marking

The *offline marking* algorithm works as follows. It has as input an MITL formula and an n-dimensional Boolean signal w of length r. For every sub-formula ψ of φ it computes its satisfiability signal $u_\psi = \chi^\psi(w)$ (we will use u when ψ is clear from the context). The procedure is recursive on the structure (parse tree) of the formula (see Algorithm 1). It goes down until the propositional variables whose values are determined directly by w, and then propagates values as it comes up from the recursion. We use OP_1 and OP_2 for arbitrary unary and binary logical or temporal operators. As a preparation for the incremental version, we do not pass w and u_φ as input or output parameters but rather store them in global data structures.

Most of the work in this algorithm is done by the COMBINE function which for $\varphi = \text{OP}_2(\varphi_1, \varphi_2)$ computes u_φ from the signals u_{φ_1} and u_{φ_2}. We describe how this function works for each of the operators, and for the sake of readability we omit the description of various optimizations. We have shown in Lemma 3.1 that timed until and

Algorithm 1: OFFLINEMITL

```
input : an MITL Formula $\varphi$ and signal $w$
switch $\varphi$ do
    case $p$
        $u_\varphi := w|_p$;
    end
    case $\mathrm{OP}_1(\varphi_1)$
        OFFLINEMITL $(\varphi_1)$;
        $u_\varphi :=$ COMBINE$(\mathrm{OP}_1, u_{\varphi_1})$;
    end
    case $\mathrm{OP}_2(\varphi_1, \varphi_2)$
        OFFLINEMITL $(\varphi_1)$;
        OFFLINEMITL $(\varphi_2)$;
        $u_\varphi :=$ COMBINE $(\mathrm{OP}_2, u_{\varphi_1}, u_{\varphi_2})$;
    end
end
```

since operators are redundant and consequently, in the remainder of the section it is sufficient to describe the COMBINE function for the following operators:

- Negation $\neg\varphi$ and disjunction $\varphi_1 \vee \varphi_2$
- Untimed *since* $\varphi_1 \mathcal{S} \varphi_2$ and *until* $\varphi_1 \mathcal{U} \varphi_2$
- Timed *once* $\diamondminus_I \varphi$ and *eventually* $\diamond_I \varphi$

4.1.1 Combine function for $\neg\varphi$ and $\varphi_1 \vee \varphi_2$

The negation $\varphi = \neg\varphi_1$ is simply computed with $u_\varphi :=$ COMBINE$(\neg, u_{\varphi_1})$, by changing the Boolean value of each singular point and open segment in the representation of u_{φ_1}.

For the disjunction $\varphi = \varphi_1 \vee \varphi_2$, the function $u_\varphi :=$ COMBINE$(\vee, u_{\varphi_1}, u_{\varphi_2})$ first refines the point-segment representation of the signals for the pairing $u' = u_{\varphi_1} || u_{\varphi_2}$. This way the value of both signals becomes uniform within every open segment. Then, we compute the disjunction at every point/segment, concatenating them in order to obtain u_φ. This procedure is illustrated in Figure 4.1.

4.1.2 Combine function for $\varphi_1 \mathcal{S} \varphi_2$ and $\varphi_1 \mathcal{U} \varphi_2$

We assume a finite signal $w = \dot{w}_0 \cdot w_0^{r_0} \cdots \dot{w}_k \cdot w_k^{r_k}$ of length $|w| = r_0 + \cdots + r_k = r$. We have shown in Lemma 3.4 that $p\mathcal{S}q$ operator is left continuous, meaning that the satisfaction of the operator at any singular point cannot differ from its satisfaction during the previous open segment. We have also shown in Lemma 3.6 that there is a finite number of rules that determine the value of u in open segments depending on the past observations of p and q. The combination of these two results gives us a straightforward recipe for computing $u = \chi^{p\mathcal{S}q}(w)$

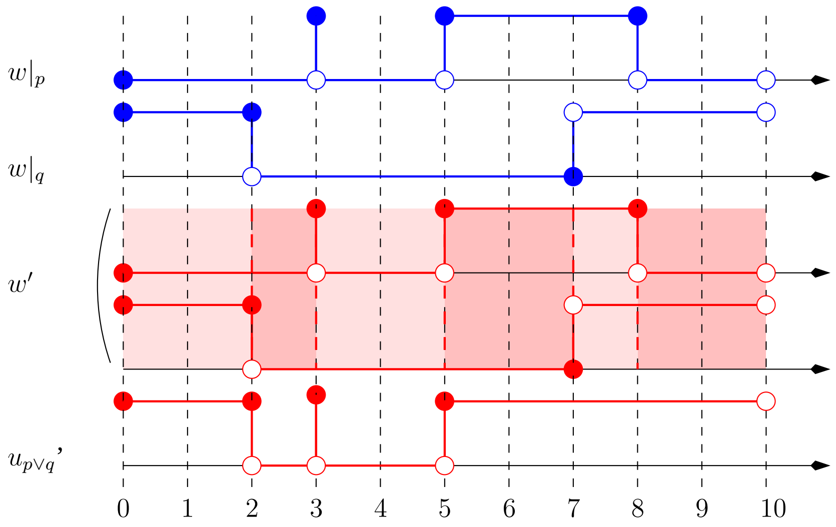


Fig. 4.1. Computing $u = \chi^{p\vee q}(w)$

Now we can describe how the function that computes the value of $u = \chi^{p\mathcal{S}q}(w)$ works. We start reading the signal w from the beginning towards the end. Following Lemma 3.4, $\dot{u}_0 = 0$, regardless of w. For every subsequent singular point, the value $\dot{u}_i$ is equal to u_{i-1}, the value of u during the previous open segment. When a new open segment of w_i is read, the procedure applies the rules of Lemma 3.6 to compute u_i, the value of u in the same segment. If p is false in w_i, then u_i is also false. Similarly, if both p and q hold in w_i, then the segment u_i is set to be true. Finally, if p holds during w_i and q is false throughout the same segment, there are three possibilities: 1) either both p and q were false at the previous singular point $\dot{w}_i$ and then u_i is set to be false; 2) q was true at $\dot{w}_i$ so u_i is set to true or 3) p was true and q false at $\dot{w}_i$ and u_i has the same value as in the previous singular point $\dot{u}_i$.

Computing the COMBINE function for $p\mathcal{U}q$ operator is symmetric to the $p\mathcal{S}q$ case. We have shown in Lemma 3.5 that *until* is right continuous, meaning that the satisfaction of the operator at any singular point is identical to its satisfaction during the next open segment. In Lemma 3.7 we provided a finite number of rules to determine the value of u in the open segments depending on the future observations of p and q. The combination of these two results provide rules for computing $u = \chi^{p\mathcal{U}q}(w)$

The computation of $u = \chi^{p\mathcal{U}q}(w)$ works as follows. The signal w is read from the end towards the beginning. We determine the value of every open segment u_i according to the rules of Lemma 3.7. If p is false in w_i, then u_i is also false. Similarly, if both p and q hold in w_i, then the value of the segment u_i is set to be true. For segments w_i where p is true and q is false, there are four possibilities: 1) w_i is the last open segment in w and u_i is false; 2) both p and q are false in $\dot{w}_{i+1}$ and u_i is set to false; 3) q is true at $\dot{w}_{i+1}$ so u_i is also set to true or 4) p is true and q false at $\dot{w}_{i+1}$ and u_i has the same value as in the next singular point $\dot{u}_{i+1}$. Every singular point $\dot{u}_i$ is set to the value of the succeeding open segment u_i, as shown by Lemma 3.5.

Example

Consider the signal from Figure 4.2-(a,b)

$$w = \dot{w}_0 \cdot w_0^2 \cdot \dot{w}_1 \cdot w_1^4 \cdot \dot{w}_2 \cdot w_2^1 \cdot \dot{w}_3 \cdot w_3^3 = \dot{\begin{pmatrix} p \\ \overline{q} \end{pmatrix}} \cdot \begin{pmatrix} p \\ \overline{q} \end{pmatrix}^2 \cdot \dot{\begin{pmatrix} \overline{p} \\ \overline{q} \end{pmatrix}} \cdot \begin{pmatrix} p \\ q \end{pmatrix}^4 \cdot \dot{\begin{pmatrix} p \\ q \end{pmatrix}} \cdot \begin{pmatrix} \overline{p} \\ q \end{pmatrix}^1 \cdot \dot{\begin{pmatrix} \overline{p} \\ q \end{pmatrix}} \cdot \begin{pmatrix} p \\ \overline{q} \end{pmatrix}^3$$

The signal $u = \chi^{p \mathcal{S} q}(w)$ is of the form $u = \dot{u}_0 \cdot u_0^2 \cdot \dot{u}_1 \cdot u_1^4 \cdot \dot{u}_2 \cdot u_2^1 \cdot \dot{u}_3 \cdot u_3^3$ and is computed with following steps:

- $\dot{u}_0$ is trivially false (Lemma 3.4)
- $u_0 = 0$ because $w_0 = \binom{p}{\overline{q}}$, $\dot{w}_0 = \binom{p}{\overline{q}}$ and $\dot{u}_0 = 0$ (case 3 (c) of Lemma 3.6)
- $\dot{u}_1 = u_0 = 0$ (Lemma 3.4)
- $u_1 = 1$ because $w_1 = \binom{p}{q}$ (case 2 of Lemma 3.6)
- $\dot{u}_2 = u_1 = 1$ (Lemma 3.4)
- $u_2 = 0$ because $w_2 = \binom{\overline{p}}{q}$ (case 1 of Lemma 3.6)
- $\dot{u}_3 = u_2 = 0$ (Lemma 3.4)
- $u_3 = 1$ because $w_3 = \binom{p}{\overline{q}}$ and $\dot{w}_3 = \binom{\overline{p}}{q}$ (case 3 (b) of Lemma 3.6)

The resulting signal $u = \dot{0} \cdot 0^2 \cdot \dot{0} \cdot 1^4 \cdot \dot{1} \cdot 0^1 \cdot \dot{0} \cdot 1^3$ is shown in Figure 4.2-(c).

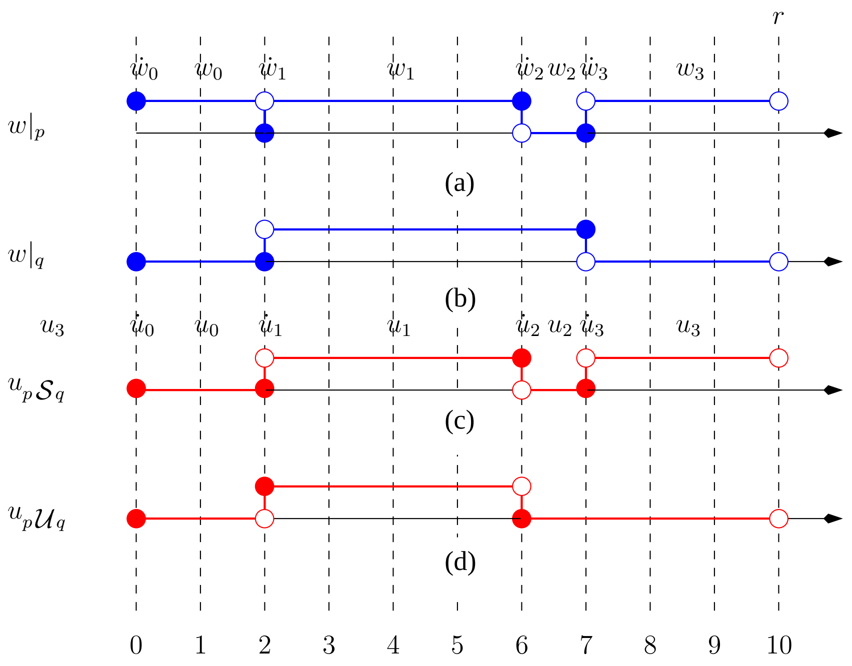
For the same input signal w, we show how $u = \chi^{p \mathcal{U} q}(w)$ is computed. Similarly to the previous case, u is of the form $u = \dot{u}_0 \cdot u_0^2 \cdot \dot{u}_1 \cdot u_1^4 \cdot \dot{u}_2 \cdot u_2^1 \cdot \dot{u}_3 \cdot u_3^3$, but now we scan w from its end to the beginning (right to left):

- $u_3 = 0$ because $w_3^3 = \binom{p}{\overline{q}}$ and it is the last segment in the input signal (finitary interpretation of until)
- $\dot{u}_3 = u_3 = 0$ (Lemma 3.5)
- $u_2 = 0$ because $w_2 = \binom{\overline{p}}{q}$ (case 1 of Lemma 3.7)
- $\dot{u}_2 = u_2 = 0$ (Lemma 3.5)
- $u_1 = 1$ because $w_1 = \binom{p}{q}$ (case 2 of Lemma 3.7)
- $\dot{u}_1 = u_1 = 1$ (Lemma 3.5)
- $u_0 = 0$ because $w_0 = \binom{p}{\overline{q}}$ and $\dot{w}_1 = \binom{\overline{p}}{\overline{q}}$ (case 3 (a) of Lemma 3.7)
- $\dot{u}_0 = u_0 = 0$ (Lemma 3.5)

The resulting signal is $u = \dot{0} \cdot 0^2 \cdot \dot{1} \cdot 1^4 \cdot \dot{0} \cdot 0^1 \cdot \dot{0} \cdot 0^3$ and by merging stationary points with the adjacent open segments, we obtain the signal u with its coarsest time partition $u = \dot{0} \cdot 0^2 \cdot \dot{1} \cdot 1^4 \cdot \dot{0} \cdot 0^4$, shown in Figure 4.2-(d).

4.1.3 Combine function for $\diamondsuit_I \varphi$ and $\diamondminus_I \varphi$

To compute $u = \chi^{\diamondsuit_I \varphi}(u_\varphi)$ and $u = \chi^{\diamondminus_I \varphi}(u_\varphi)$ we first observe that whenever φ holds in an interval J, u holds in the interval $J \ominus I \cap \mathbb{T}$ (respectively $J \oplus I \cap \mathbb{T}$). Hence, the essence of the procedure is to "propagate" the intervals in u_φ where φ holds either forward or backward. We employ the auxiliary concept of *interval covering* of a signal.



r
w_0 w_0 w_1 w_1 w_2 w_2 w_3 w_3
$w|_p$
(a)
$w|_q$
(b)
u_3 u_0 u_0 u_1 u_1 u_2 u_2 u_3 u_3
$u_p\mathcal{S}_q$
(c)
$u_p\mathcal{U}_q$
(d)
0 1 2 3 4 5 6 7 8 9 10

Fig. 4.2. Computing $u = \chi^{p\,\mathcal{S}_q}(w)$ and $u = \chi^{p\,\mathcal{U}_q}(w)$

Definition 4.1 (Interval covering). *For a signal w of finite length defined over $\mathbb{T} = [0,r)$, its interval covering is a sequence $\mathcal{I}_w = I_0,\ldots,I_k$ such that $\bigcup I_i = \mathbb{T}$ and $I_i \cap I_j = \emptyset$ for any $i \neq j$. An interval covering is said to be consistent with a finite length signal w if $w[t] = w[t']$ for every t,t' that belong to the same interval $I_i \in \mathcal{I}_w$. We denote by $\mathcal{I}_w$ the minimal interval covering consistent with the signal w. The set of positive intervals in $\mathcal{I}_w$ is $\mathcal{I}_w^+ = \{I \in \mathcal{I}_w | \forall t \in I, w[t] = 1\}$ and the set of negative intervals is $\mathcal{I}_w^- = \mathcal{I}_w - \mathcal{I}_w^+$.*

Let us assume that $\mathcal{I}_{u_\varphi}$ is the minimal interval covering consistent with u_φ. Then $u = \chi^{\Diamond_I \varphi}(u_\varphi)$ is computed using the following procedure. For every positive interval $I^+ \in \mathcal{I}_\varphi^+$, we compute its back shifting (Minkowski difference saturated by $\mathbb{T}$) $I^+ \ominus I \cap \mathbb{T}$ and insert it to $\mathcal{I}_u^+$. This set represents the intervals where $\Diamond_I \varphi$ is satisfied, and the property is violated outside these intervals. Overlapping positive intervals in $\mathcal{I}_u^+$ are merged to obtain the minimal interval covering[1] of u.

The combine function for $u = \chi^{\Diamond\!\!\!-_I \varphi}(u_\varphi)$ is computed in a similar way. For every positive interval $I^+ \in \mathcal{I}_\varphi^+$, we compute its forward shifting (Minkowski sum saturated by $\mathbb{T}$) $I^+ \oplus I \cap \mathbb{T}$ and insert it to $\mathcal{I}_u^+$, and merge the overlapping positive intervals in $\mathcal{I}_u^+$ to obtain the minimal interval covering of u.

[1] Note that the similar operation can be applied to negative intervals in $\mathcal{I}_\varphi^-$, in order to directly compute intervals where $\Diamond_I \varphi$ is violated. This procedure is not necessary for offline monitoring, but is useful for the incremental version of the algorithm

Example

We consider the signal $w|_p = \dot{0}\cdot 0^3 \cdot \dot{1}\cdot 0^2 \cdot \dot{1}\cdot 1^1 \cdot \dot{1}\cdot 0^{0.5}\cdot \dot{1}\cdot 1^{3.5}$. The minimal interval covering consistent with $w|_p$ is the sequence $\mathcal{I}_p = [0,3), [3,3], (3,5), [5,6], (6,6.5), [6.5,10)$, the set of positive intervals in $\mathcal{I}_p$ is $\mathcal{I}_p^+ = \{[3,3],[5,6],[6.5,10)\}$ and the set of negative intervals in $\mathcal{I}_p$ is $\mathcal{I}_p^- = \{[0,3),(3,5),(6,6.5)\}$.

The COMBINE function that generates the signal $u = \chi^{\Diamond_{[1,2]}}(w)$ is computed with following steps:

- The Minkowski difference $I^+ \ominus [1,2] \cap [0,10)$ is computed for every positive interval $I^+ \in \mathcal{I}_p^+$ and the resulting interval is inserted into the set $\mathcal{I}_u^+$. After applying this operation, $\mathcal{I}_u^+ = \{[1,2],[3,5],[4.5,9)\}$. After merging the overlapping positive intervals we obtain $\mathcal{I}_u^+ = \{[1,2],[3,9)\}$.

The resulting signal is $u = \dot{0}\cdot 0^1 \cdot \dot{1}\cdot 1^1 \cdot \dot{1}\cdot 0^1 \cdot \dot{1}\cdot 1^6 \cdot \dot{0}\cdot 0^1$ as shown in Figure 4.5-(b). The COMBINE function that generates the signal $u = \chi^{\diamondminus_{[1,2]}}(w)$ is computed with following steps:

- The Minkowski sum $I^+ \oplus [1,2] \cap [0,10)$ is computed for every positive interval $I^+ \in \mathcal{I}_p^+$ and the resulting interval is inserted into the set $\mathcal{I}_u^+$. After applying this operation, $\mathcal{I}_u^+ = \{[4,5],[6,8],[7.5,10)\}$. After merging the overlapping positive intervals we obtain $\mathcal{I}_u^+ = \{[4,5],[6,10)\}$.

The resulting signal is $u = \dot{0}\cdot 0^4 \cdot \dot{1}\cdot 1^1 \cdot \dot{1}\cdot 0^1 \cdot \dot{1}\cdot 1^4$ as shown in Figure 4.5-(d).

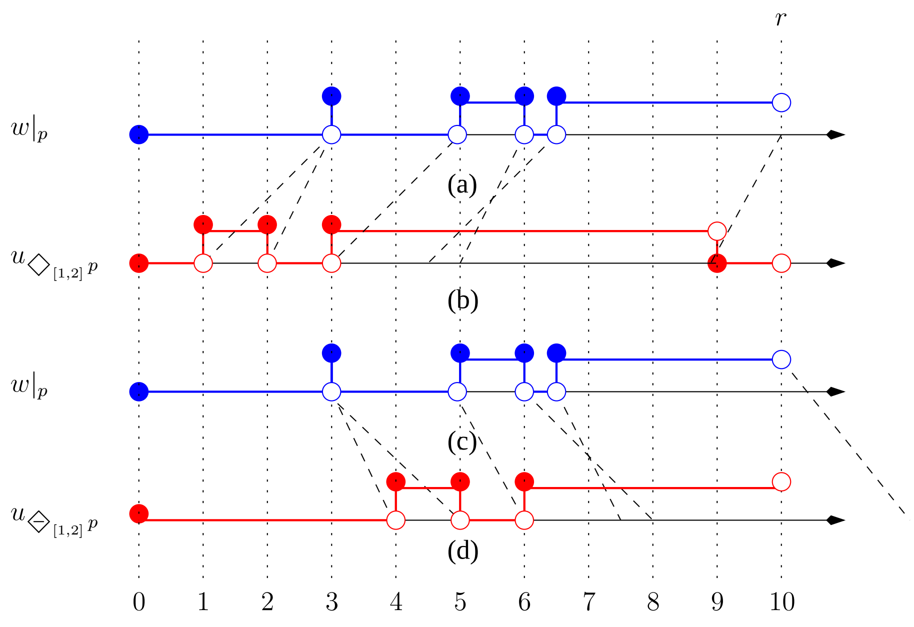


Fig. 4.3. Computing $u = \chi^{\Diamond_{[1,2]}\, p}(w)$ and $u = \chi^{\diamondminus_{[1,2]}\, p}(w)$

4.2 Incremental Marking

This approach combines the simplicity of the offline procedure with the advantages of online monitoring in terms of early detection of violation or satisfaction and typically smaller memory requirements. After observing a prefix $w[0, t_1]$ of the signal we apply the offline procedure (without applying the finitary interpretation rules for future temporal operators, these are applied only at the end of the input trace). If, as a result, $u_\varphi = \chi^\varphi(w)$ is determined at time 0 we are done. Otherwise, we observe a new segment $w[t_1, t_2]$ and then apply the same procedure based on $w[0, t_2]$.

A more efficient implementation of this procedure need not start the computation from scratch each time a new segment is observed. It will be often the case that $u_\psi = \chi^\psi(w)$ for some sub-formulae ψ is already determined for some subset of $[0, t_1]$, based on $w[0, t_1]$. In this case we only need to propagate upwards the new information obtained from $w[t_1, t_2]$, combined possibly, with some residual information from the previous segments that was not sufficient for determination of the satisfaction of the super formula. The choice of granularity (lengths of segments) in which this procedure is invoked depends on trade-offs between the computational cost and the importance of early detection.

The essence of the incremental marking procedure lies in the observation that the evaluation of a Boolean or future temporal formula φ at time t, depends on the values of their subformulae at $t' \geq t$. This implies that if u_φ is already determined at some interval $[0, t_1]$, we only need to keep the values of the satisfaction signal of its subformulae after t_1. Similarly, a past temporal operator ψ depends on the satisfaction of its subformulae at $t' \leq t$. The algorithm needs minor (and symmetric) adaptations between incremental marking for future and past temporal operators, and in the remaining of the section we focus on the procedure dealing with future temporal formulae.

Incremental marking is performed using a kind of piecewise-online procedure invoked each time a new segment of w, denoted by Δ_w, is observed. For each sub-formula ψ the algorithm stores its already-computed satisfaction signal partitioned into a concatenation of two signals $u_\psi \cdot \Delta_\psi$ with u_ψ consisting of values already propagated to the super-formula of ψ, and Δ_ψ consists of values that have already been computed but which have not yet been propagated to the super-formula and can still influence its satisfaction.

Initially all signals are empty. Each time a new segment Δ_w is read, a recursive procedure similar to the offline procedure is invoked, which updates every u_ψ and Δ_ψ from the bottom up. The difference with respect to the offline algorithm is that only the segments of the signal that have not been propagated upwards participate in the update of their super-formulae. This may result in a lot of saving when the signal is very long (the empirical demonstration of this claim is given in section 7.1.2).

As an illustration consider $\varphi = \textsc{op}(\varphi_1, \varphi_2)$ and the corresponding truth signals of Figure 4.4-(a). Before the update we always have $|u_\varphi \cdot \Delta_\varphi| = |u_{\varphi_1}| = |u_{\varphi_2}|$: the parts Δ_{φ_1} and Δ_{φ_2} that may still affect φ are those that start at the point from which the satisfaction of φ is still unknown. We apply the COMBINE procedure on Δ_{φ_1} and Δ_{φ_2} to obtain a new (possibly empty) segment α of Δ_φ. This segment is appended to Δ_φ in order to be propagated upwards, but before that we need to shift the borderline between u_{φ_1} and Δ_{φ_1} (as well as between u_{φ_2} and Δ_{φ_2}) in order to reflect the update of Δ_φ. The procedure is described in Algorithm 2.

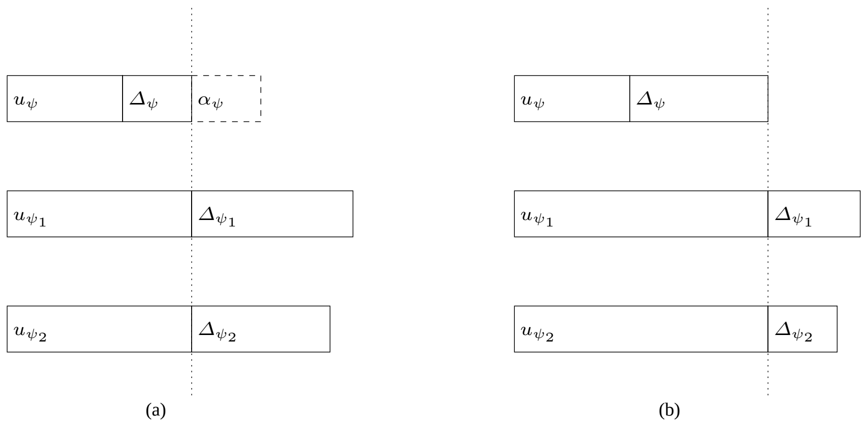


Fig. 4.4. A step in an incremental update: (a) A new segment α for ψ is computed from Δ_{ψ_1} and Δ_{ψ_2}; (b) α is appended to Δ_ψ and the endpoints of u_{ψ_1} and u_{ψ_1} are shifted forward accordingly.

Example

We illustrate the incremental monitoring procedure on the MITL formula $\varphi = \Box(p \rightarrow \Diamond_{[1,2]}\, q)$. The input signal w is split into three segments Δ^1_w, Δ^2_w and Δ^3_w and the incremental marking procedure is applied upon the arrival of each such segment:

1. The first step of the monitoring procedure is computed when the first segment $\Delta^1_w = \dot{\binom{\overline{p}}{q}} \cdot \binom{\overline{p}}{q}^2 \cdot \dot{\binom{p}{q}} \cdot \binom{p}{q}^{0.5} \cdot \dot{\binom{p}{\overline{q}}} \cdot \binom{p}{\overline{q}}^{1.5}$ is appended to w. After applying recursively the marking procedure and computing u_ψ for the subformulae ψ of φ. Figure 4.5-(a) shows the computed signals and as we can see, u_φ for the top level formula φ remains empty. Note that the segment of w defined over $[0, 2)$ as well as the entire computed segment of $u_{\Diamond_{[1,2]}}\, q$ can be discarded, since they do not affect any more the satisfaction of their corresponding super-formulae.
2. The segment $\Delta^2_w = \dot{\binom{\overline{p}}{q}} \cdot \binom{\overline{p}}{q}^3 \cdot \dot{\binom{p}{q}} \cdot \binom{p}{q}^{0.5} \cdot \dot{\binom{p}{\overline{q}}} \cdot \binom{p}{\overline{q}}^{0.5}$ is appended to the previous segment of w, and the incremental marking procedure is applied again, computing new segments of satisfaction signals for sub-formulae of φ. The satisfaction of the top formula remains undetermined. The satisfaction signals for subformulae of φ after applying the marking procedure are shown in Figure 4.5-(b).
3. Finally, the third segment $\Delta^2_w = \dot{\binom{\overline{p}}{\overline{q}}} \cdot \binom{\overline{p}}{\overline{q}}^2$ is appended to w and the incremental marking procedure is applied again. Now, all the subformulae of φ, including the top level formula itself can be updated, and since u_φ is false at $t = 0$ (see Figure 4.5-(c)), we can conclude that the formulae is violated by w and stop the procedure.

Algorithm 2: INC-OFFLINE-MITL

```
input : an MITL Formula $\varphi$ and an increment $\Delta_w$ of a signal
switch $\varphi$ do
    case $p$
        $\Delta_\varphi := \Delta_\varphi \cdot w_p(\Delta_w)$;
    end
    case $\textsc{op}_1(\varphi_1)$
        INC-OFFLINE-MITL $(\varphi_1)$;
        $\alpha := \textsc{Combine}(\textsc{op}_1, \Delta_{\varphi_1})$;
        $d := |\alpha|$ ;
        $\Delta_\varphi := \Delta_\varphi \cdot \alpha$ ;
        $u_{\varphi_1} := u_{\varphi_1} \cdot \langle \Delta_{\varphi_1} \rangle_d$ ;
        $\Delta_{\varphi_1} := d \backslash \Delta_{\varphi_1}$
    end
    case $\textsc{op}_2(\varphi_1, \varphi_2)$
        INC-OFFLINE-MITL $(\varphi_1)$;
        INC-OFFLINE-MITL $(\varphi_2)$;
        $\alpha := \textsc{Combine}(\textsc{op}_2, \Delta_{\varphi_1}, \Delta_{\varphi_2})$;
        $d := |\alpha|$ ;
        $\Delta_\varphi := \Delta_\varphi \cdot \alpha$ ;
        $u_{\varphi_1} := u_{\varphi_1} \cdot \langle \Delta_{\varphi_1} \rangle_d$ ;
        $\Delta_{\varphi_1} := d \backslash \Delta_{\varphi_1}$ ;
        $u_{\varphi_2} := u_{\varphi_2} \cdot \langle \Delta_{\varphi_2} \rangle_d$ ;
        $\Delta_{\varphi_2} := d \backslash \Delta_{\varphi_2}$
    end
end
```

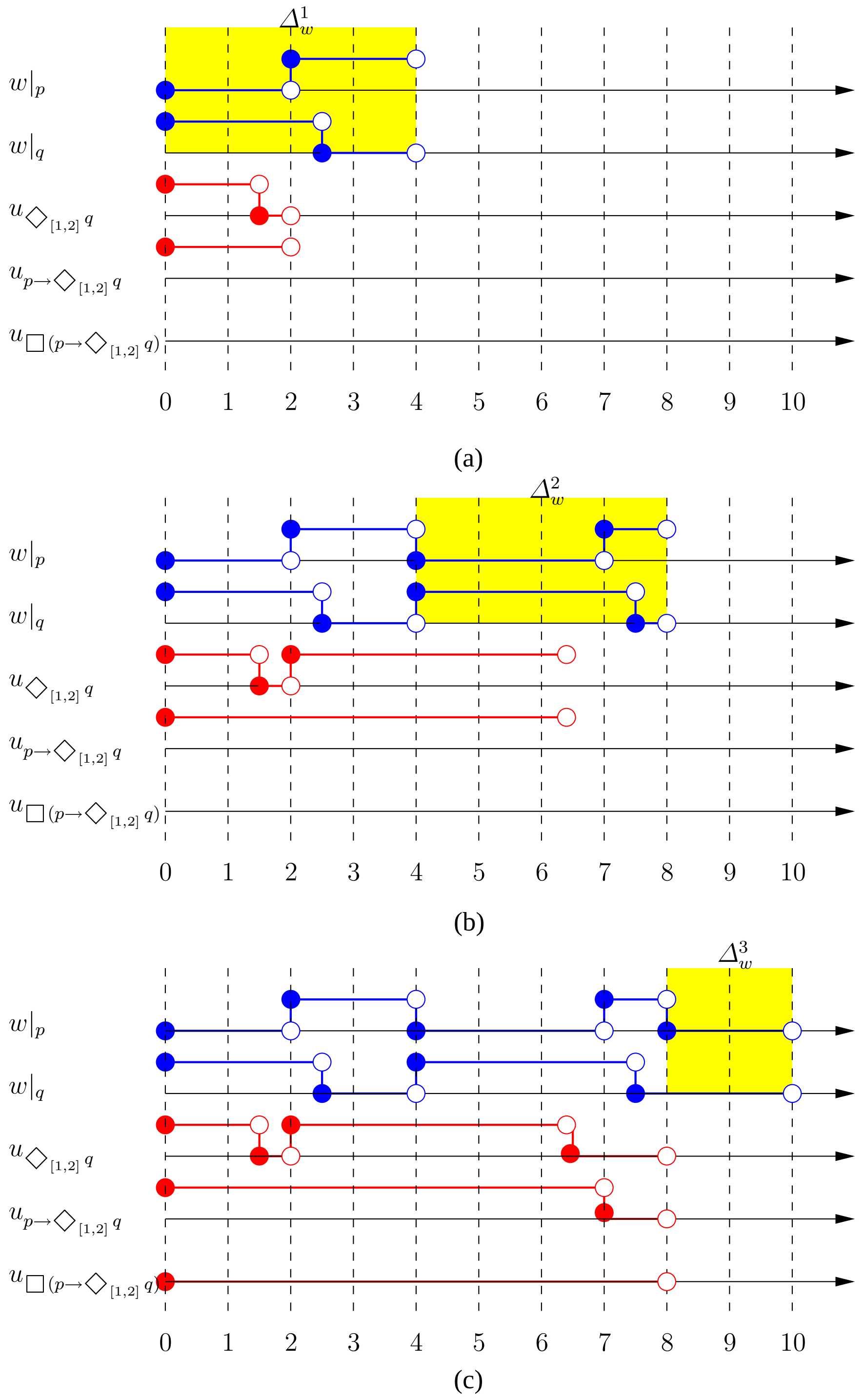


Fig. 4.5. Satisfaction signals u_ψ for sub-formulae ψ of $\varphi = \Box(p \to \Diamond_{[1,2]} q)$ computed incrementally upon receiving (a) Δ_w^1 (b) Δ_w^2 and (3) Δ_w^3

5

Monitoring Continuous Behaviors

In this chapter we extend the results of the previous section toward real-valued (continuous, analog) signals, that is, functions from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}^n$. Such signals form a much richer class of objects and the first issue to be resolved is to define the class of properties that we use. Our choice is to use properties whose checking can be transformed into checking MITL against a "Booleanization" of the signal via finitely many predicates. Once this is defined, all that remains is to handle technical problems related to the (sampled) representation of such signals inside the computer.

5.1 Signal Booleanization and the Logic STL

We consider signals of the form $\xi : \mathbb{T} \to X$ over state-space $X \subseteq \mathbb{R}^n$. A predicate over X is a function $\mu : X \to \mathbb{B}$ which can be syntactically expressed using arithmetic functions and inequalities over the state variable, for example, $x < 5$ or $|x^2 - y^2| \leq 1$. We consider a finite set of such predicates such that by applying them pointwise we obtain Boolean signals describing the evolution over time of the truth values of these predicates with respect to w.

Definition 5.1 (Booleanization). *Let $\xi : \mathbb{T} \to X$ be a real-valued signal and let $M = \{\mu_1, \ldots, \mu_m\}$ be be a set of predicates of the form $\mu_i : X \to \mathbb{B}$. The M-Booleanization of ξ, denoted by $M(\xi)$, is the signal $w : \mathbb{T} \to \mathbb{B}^m$ satisfying for every i and for every t*

$$w_i[t] = \mu_i(x_1, \ldots, x_n).$$

Events such as *rising* and *falling* in the Boolean signal correspond to some *qualitative* changes in the real-valued signal, for example threshold crossing of some continuous variable.

We now define the *signal temporal logic* STL as an extension of MITL that can express properties that depend on the Booleanization of the signal. That is, we are concerned with properties such that if two signals ξ and ξ' satisfy $M(\xi) = M(\xi')$ then for every formula φ, $\xi \models \varphi$ iff $\xi' \models \varphi$.

The syntax of STL is thus parameterized by a set of real-valued function symbols $f_1, \ldots f_k$. A term of STL is either a rational constant c, a real-valued variable x or a function $f(x_1, \ldots x_n)$. A predicate of STL is an expression of the form $E \sim c$ where E

is a term, and $\sim\in\{<,\leq,=,\geq,>\}$. The whole syntax is very much like MITL where predicates have the same role as atomic propositions:

$$\varphi := p \mid E \sim c \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \, \mathcal{U}_I \, \varphi_2 \mid \varphi_1 \, \mathcal{S}_I \, \varphi_2$$

where p belongs to a set $P=\{p_1,\ldots,p_n\}$ of propositions, E is a term, $\sim\in\{<,\leq,=,\geq,>\}$, c is a constant and I is an interval of the form $[a,b],\ [a,b),\ (a,b],\ (a,b),\ [a,\infty)$ or (a,∞) where $0\leq a<b$ are rational numbers.

Example

An example of a property that can be expressed in STL is a mixed signal stabilization property that has the following requirements:

- The absolute value of a continuous signal x is always less than 6
- When the (Boolean) trigger rises, within 600 time units $|x|$ has to drop below 1 and stay like that for at least 300 time units

This property is illustrated in Figure 5.1 and expressed in STL as:

$$\Box(|x|<6 \wedge (\uparrow trigger \rightarrow \Diamond_{[0,600]} \Box_{[0,300]}(|x|<1)))$$

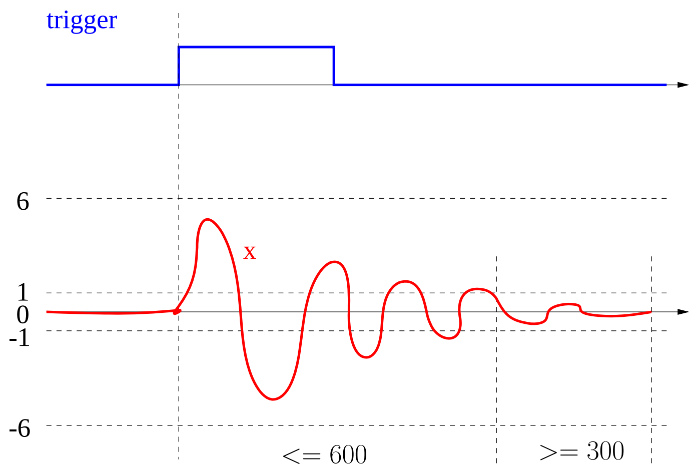


Fig. 5.1. Mixed signal stabilization property

The monitoring of STL can be easily reduced to Booleanization and monitoring against the MITL-skeleton of the formula.

5.2 Continuous Signals and their Representation

The previous section dealt with the *definition* of the satisfaction of a property by a real-valued signal. However, to really implement a monitoring procedure, we have to cope with some technical problems related to the computer representation of continuous signals.

As we have seen in section 3.1, finite non-Zeno Boolean signals, albeit the fact that they are defined over dense time domain, admit an *exact finite representation* via the switching (singular) points and the open segments that define their *true* and *false* intervals. This is no longer the case for continuous signals where we have a contrast between the *ideal mathematical object*, consisting of an uncountable number of pairs $(t, \xi[t])$ with t ranging over some interval $[0, r) \subseteq \mathbb{T}\}$, and any *finite* representation which consist of a collection of such pairs, with t restricted to range over a finite set of *sampling points*.

The values of ξ at sampling points t_1 and t_2 do not determine the values of ξ inside the interval (t_1, t_2). They may, at most, impose some constraints on these values. Such constraints can be based on the dynamics of the generating system and on the manner in which the numerical simulator produces the signal values at the sampling points. Numerical analysis is a very mature domain with a lot of accumulated experience concerning tradeoffs between accuracy and computation time. Its major premise is that given a model of the system as a continuous dynamical system defined by a differential equation[1], one can improve the quality of a discrete-time approximation of its behavior by employing denser sets of sampling points and more sophisticated numerical integration procedures.[2]

In order to speak quantitatively about the approximation of a signal by another we need the concept of a *distance/metric* imposed on the space of continuous signals. A metric is a function that assigns to two signals ξ_1 and ξ_2 a non-negative value $\rho(\xi_1, \xi_2)$ which indicates how they resemble each other. Using metrics one can express the "convergence" of a numerical integration scheme as the condition that $\lim_{d \to 0} \rho(\xi, \xi_d) = 0$ where ξ is the ideal mathematical signal and ξ_d is its numerical approximation using an integration step d.

Metrics and norms for continuous signals are used extensively in circuit design, control and signal processing. There are, however, major problems concerned with their application to property monitoring due to the incompatibility between the *continuous* nature of the signals and the discrete nature of their Booleanization, a phenomenon which is best illustrated using the following simple example. Consider the property $\Box(x > 0)$ and an ideal mathematical signal ξ that satisfies the property but which passes very close to zero at some points. We can easily deform ξ into a signal ξ' which is *very close* to ξ under any reasonable continuous metric, but according to the metric induced by the property, these signals are as distant as can be: one of them satisfies the property and the other violates it (see Figure 5.2).

Moreover, if the sojourn time of a signal below zero is short, an arbitrary shift in the sampling can make the monitor miss the zero-crossing event and declare the signal as satisfying (see Figure 5.3). In this sense properties are not *robust* as small variations in the signal may lead to large variations in its property satisfaction. Let us mention some interesting ideas [KC06] concerning new metrics for bridging the gap between the

[1] It is worth noting that some models used for rapid simulation of transistor networks cannot always be viewed as continuous dynamical systems in the classical mathematical sense.

[2] For systems which are stable the quality can be improved indefinitely.

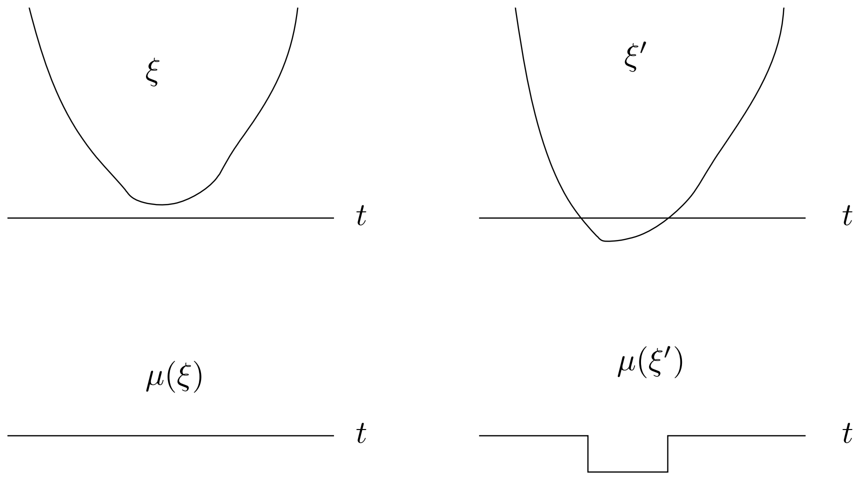


Fig. 5.2. Two signals which are close from a continuous point of view, one satisfying the property $\Box(x > 0)$ and one violating it.

continuous and the discrete points of view. Such metrics are expressible, by the way, in STL [NM07].

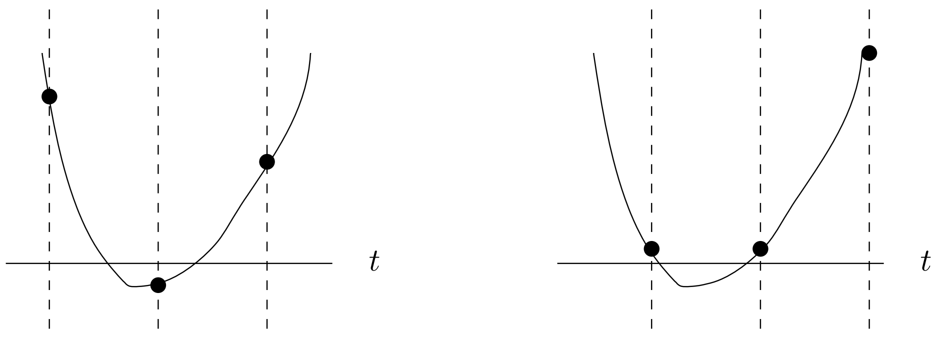


Fig. 5.3. Shifting the sampling points, zero crossing can be missed.

We handle the abovementioned issues pragmatically. The following assumptions facilitate the monitoring of sampled continuous signals against STL properties, passing through Booleanization:

1. *Sufficiently-dense sampling*: the simulator detects every change in the truth value of any of the predicates appearing in the formula at a sufficient accuracy. This way the positive intervals of all the Boolean signals that correspond to these predicates are determined. This requirement imposes some level of sophistication on the simulator that has to perform several back-and-forth iterations to locate the time instances where a threshold crossing occurs. Many simulation tools used in industry have already such event-detection features. For instance, VERILOG-AMS [Acc08] provides event-detection feature using constructs such as `@cross`, `@last_crossing` or `@above` which allow to detect the crossings of thresholds with arbitrary precision,

by forcing the simulator to make smaller time steps around the defined threshold. A survey of the treatment of discontinuous phenomena by numerical simulators can be found in [Mos99].

2. *Bounded variability*: some restrictive assumptions can be made about the values of the signal between two sampling points t_1 and t_2. For example one may assume that ξ is monotone so that if $\xi[t_1] \leq \xi[t_2]$ then $\xi[t_1'] \leq \xi[t_2']$ for every t_1' and t_2' such that $t_1 < t_1' < t_2' < t_2$. An alternative condition could be a condition a-la Lipschitz: $|\xi[t_2] - \xi[t_1]| \leq K|t_2 - t_1|$. Such conditions guarantee that the signal does not get wild between the sampling points, otherwise property checking based on these values may become useless.

Under such assumptions every continuous signal given by a discrete-time representation, based on sufficiently-dense sampling, induces a well-defined Boolean signal ready for MITL monitoring. When we don't have direct connection with the simulator as in the case with the AMT tool developed during this thesis, we replace the hypothesis of sufficiently-dense sampling by *interpolation*. That is, when we have two consecutive sampling points $t_1 < t_2$ such that one satisfies a predicate and the other does not, we use linear interpolation to "compute" the value of the signal throughout the interval (t_1, t_2) and detect the singular point t' where the value of the predicate changes. The procedure is illustrated in Figure 5.4

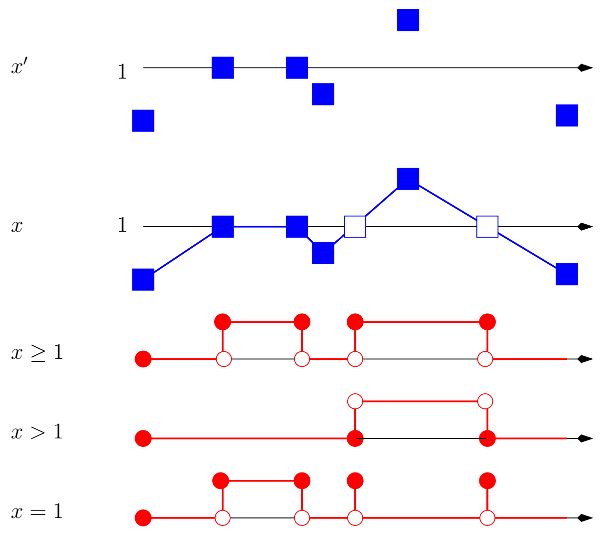


Fig. 5.4. Transformation of a continuous signal to its Boolean abstraction via interpolation and numerical predicates. The signal indicated by x' was not sufficiently dense with respect to the predicates $x \sim 1$ and hence two additional sampling points were added.

5.3 Discussion

The standards of exactness and exhaustiveness as maintained in discrete verification cannot and should not be exported to the continuous domain. While one can steer the analog simulator to make sufficient samplings around points of interest in the generated signals, the absolute precision cannot be achieved. However, the simulator can be guided to detect threshold crossings with some arbitrary tolerance, and even if we are not guaranteed that all events are detected, we can compensate for that by using safety margins in the predicates and properties. Note that the problem of precision achieved by analog simulators is more general than in the context of property-based monitoring and concerns all the validation techniques.

6

Analog Monitoring Tool

In this chapter, we present the *analog monitoring tool* AMT that implements the algorithms for monitoring timed and continuous behaviors described in Chapters 4 and 5. AMT is a stand-alone tool with a graphical user interface written in C++ for GNU/Debian Linux x86 machines. The user interface is based on the library QT[1], while QWT[2] was used for visualizing results (plots for Boolean and continuous signals). Figure 6.1 shows the general architecture of AMT. The user has to translate an informal specification (usually written in textual form) into an STL/PSL property, which is just syntactic sugar around STL and will be presented in more detail section 6.1.1. The tool takes as inputs the STL/PSL specification and a set of simulation traces. The specification is parsed and transformed into a property checker that monitors whether the simulation traces satisfy the property and outputs the result.

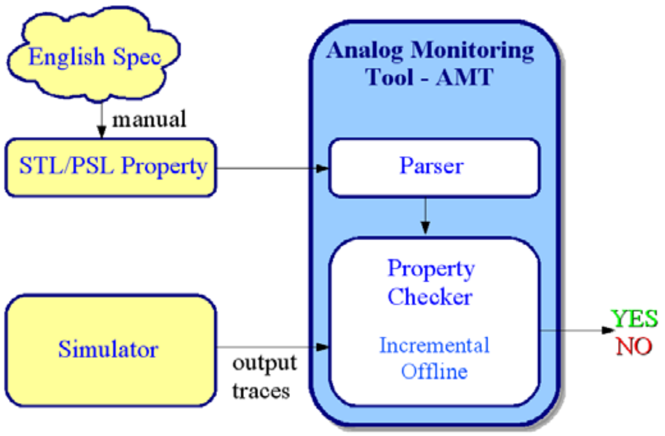
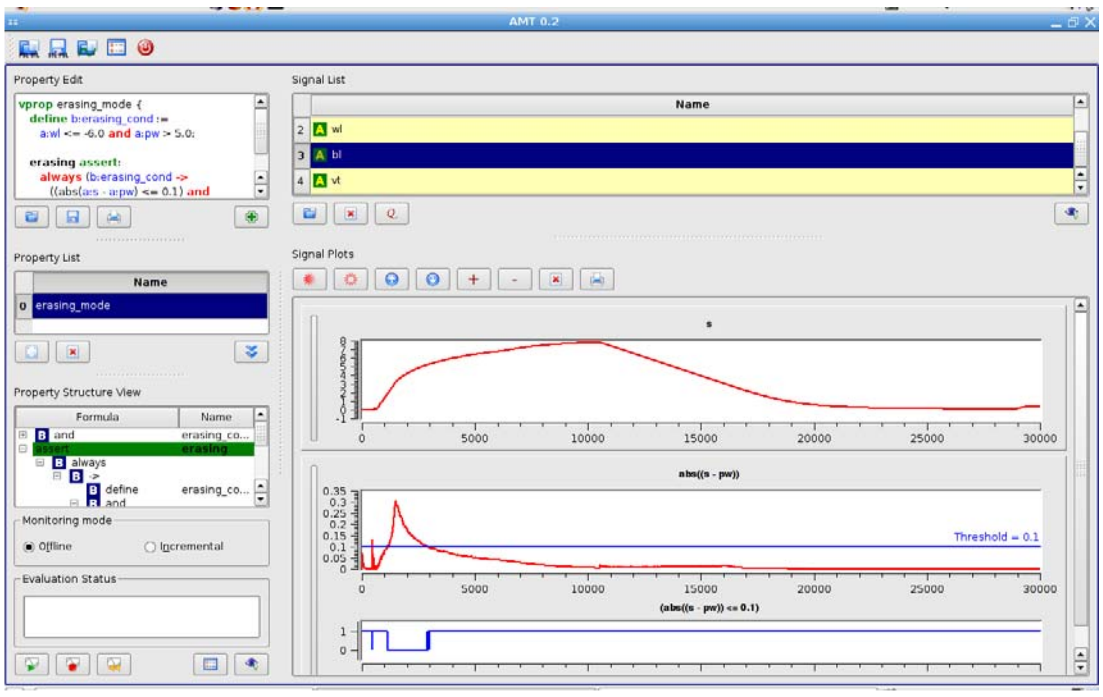


Fig. 6.1. Architecture of the AMT tool

The main window of the application is partitioned into five frames that allow the user to manage STL/PSL properties and input signals, evaluate the correctness of the simulation traces with respect to a specification and finally visualize the results. The **property**

[1] http://www.trolltech.com
[2] http://qwt.sourceforge.net

edit frame contains a text editor for writing, importing and exporting STL/PSL specifications, which are then translated into an internal data structure based on the parse-tree of the formula stored in the **property list** frame. An STL/PSL specification is imported into the **property evaluation** frame for its monitoring with respect to a set of input simulation traces, in either *offline* or *incremental* modes. The static import of the input traces is done via the **signal list** frame. The imported input signals, as well as signals associated to the subformulae of a specification can be visualized by the user from the **signal plots** frame. A screenshot of the main window is shown in Figure 6.2.


```
vprop erasing_mode {
  define b:erasing_cond :=
    a:wl <= -6.0 and a:pw > 5.0;

  erasing assert:
    always (b:erasing_cond ->
      ((abs(a:s - a:pw) <= 0.1) and
```

	Name
2	wl
3	bl
4	vt

	Name
0	erasing_mode

Formula	Name
and	erasing_co...
assert	erasing
always	
->	
define	erasing_co...
and	

Monitoring mode: [✓ Offline] Incremental

Fig. 6.2. AMT main window

6.1 Property Management

The specifications in AMT are written in a simple editor with syntax highlighting for the STL/PSL language described below. An STL/PSL specification is then transformed into a structure adapted for the monitoring purpose, following the parse-tree of the formula. The user can hold more than one specification that is ready for evaluation in the property list frame.

6.1.1 Property Format

AMT tool supports the STL/PSL language, which provides syntactic sugar to STL and is inspired by the PSL language with additional constructs intended to simplify the process of property specification. Each top-level STL/PSL property is declared as an *assertion*, and a number of assertions can be grouped into a single logical unit in order to monitor them simultaneously. We also add a definition directive which allows the user to declare a formula and give it a name, and then refer to it as a variable within the assertions. The syntax of STL/PSL is defined with the following production rules

```
varphi :==
  vprop NAME {
    { define_directive } { assert_directive }
}

define_directive :==
  define b:NAME := varphi
  | define a:NAME := phi

assert_directive :==
  NAME assert : varphi
```

where `varphi` corresponds to a temporal property and `phi` to an analog operation. The set of operators that are included in STL/PSL is summarized in Tables 6.1 and 6.2.

6.1.2 Property Evaluation

The correctness of an STL/PSL specification with respect to input traces is monitored through the property evaluation frame. The frame shows the set of assertions in a tree view, following the parse structure of the formula. The user can choose between *offline* and *incremental* evaluation of the specification.

In the offline case, the input signals are fetched from the signal list frame and the assertions are checked with respect to them. If one or more signals are missing, the monitoring procedure still tries to evaluate the property, but without guaranteeing a conclusive result.

For the incremental procedure, AMT acts as a server that waits for a connection from the simulator. Once the connection is established, the simulator sends input segments incrementally. The monitor alternates between reception of new input segments and incremental evaluation of the assertions. The user can configure the following parameters for the incremental evaluation:

- The user can set the TCP/IP port on which the tool and the simulator will communicate
- *timeout* value that defines the period between two consecutive evaluations. Simulations of analog circuit often have tens or even hundreds of thousands of samples per signal. Hence, it is usually not wise to re-evaluate the property upon receiving every new individual sample. This option enables to accumulate input received from the simulator between any two periods defined by the timeout value and apply the incremental checking procedure only at the instants when the timer expires.

There is no pre-defined optimal value for the timeout, and it represents a trade-off between the frequency of computations and the possibility of earlier detection of violation/satisfaction of a property
- The incremental procedure often provides better memory management that the offline one, because the parts of the signals that have been fully determined and are not needed by their super-formulae can be discarded. However, in some situations, one would prefer to keep the entire signal for visualization and debugging purposes. The tool allows the user to choose through the “keep history” option whether the entire signal is kept, or only its segments that are needed for subsequent evaluations

There are three manners to end the incremental monitoring procedure:

1. All assertions become determined and AMT stops the evaluation closing the connection with the simulator;
2. The special termination packet is received from the simulator indicating the end of the input traces. In that case the tool completes the evaluation of assertions with respect to the finitary semantics of the specification language operators;
3. The user explicitly stops the procedure before the end of simulation via the GUI (reset button). In that case the connection with the simulator is closed and the evaluation remains undetermined;

AMT shows visually the evaluation result of an assertion, choosing a different color scheme for *undetermined*, *satisfied* and *violated* assertions. Each subformula of the specification has an associated signal with it, which can be visualized within the signal plots frame. The visualization of the associated signals can be used for understanding why an assertion holds/fails. During the incremental evaluation, if the “keep history” option is enabled, all the signals within the signal plots frame are updated in real-time as new results are computed.

6.2 Signal Management

The signals in AMT can be either real-valued or Boolean. Signals are input traces that can be imported into the tool in an offline or incremental fashion. But signals are also associated to each subformula of an STL/PSL specification. The user can visualize them from the signal plots frame.

Offline Signal Input

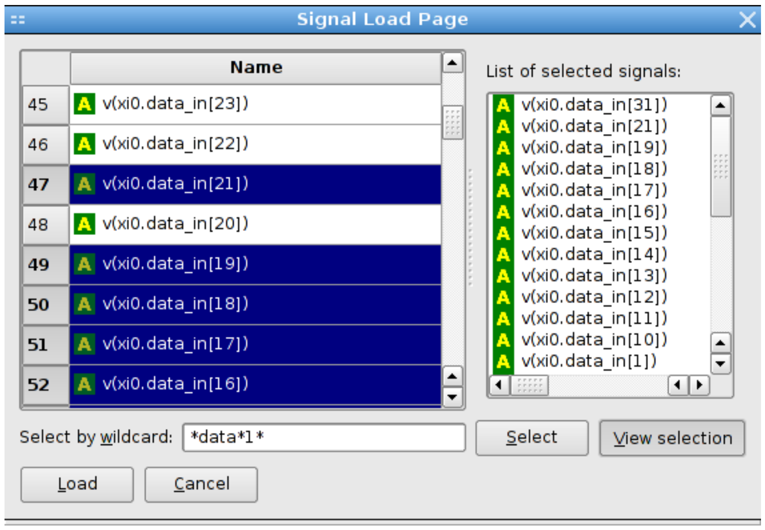
Signals can be statically loaded from the signal list frame. AMT currently supports the following input formats:

out The output format of the NANOSIM [Nan08] simulator. The *current* and *voltage* signals are loaded, while *logical* signals are ignored.

vcd The subset of Value Change Dump [Iee01] file format including real and 2-valued Boolean signals, commonly used for dumping simulations.

raw The Berkeley Spice binary and ASCII file format for simulation dumps.
txt This is a simple Ascii format that can be dumped from the CosmosScope [Syn04] wave calculator tool

The analog simulation traces are usually very large. A typical file generated by the simulation of a complex mixed or analog circuit contains hundreds of signals, and often exceeds hundreds of megabytes of data. AMT has been designed to be able to deal with very large files and has been tested with simulation dumps exceeding $2GB$ of memory. While a standard simulation file contains hundreds of signals, an STL/PSL specification usually refers only to several. Hence, there is a need to efficiently navigate through the list of available signals. For this purpose, AMT provides the option of multiple selection of signals, as well as the selection of signals by a filter. For instance, in Figure 6.3, the filter `*data*1*` selects all signals that have the pattern `data` withing their names followed (not necesseraly immediately) by `1`. Moreover, an additional window shows the list of currently selected signals.



	Name
45	v(xi0.data_in[23])
46	v(xi0.data_in[22])
47	v(xi0.data_in[21])
48	v(xi0.data_in[20])
49	v(xi0.data_in[19])
50	v(xi0.data_in[18])
51	v(xi0.data_in[17])
52	v(xi0.data_in[16])

List of selected signals:
v(xi0.data_in[31])
v(xi0.data_in[21])
v(xi0.data_in[19])
v(xi0.data_in[18])
v(xi0.data_in[17])
v(xi0.data_in[16])
v(xi0.data_in[15])
v(xi0.data_in[14])
v(xi0.data_in[13])
v(xi0.data_in[12])
v(xi0.data_in[11])
v(xi0.data_in[10])
v(xi0.data_in[1])

Select by wildcard: *data*1*

Fig. 6.3. AMT selection of signals

Incremental Signal Input

Signals can be imported incrementally to AMT, via a simple TCP/IP protocol. A simulator that produces input signals needs to connect to AMT during the *incremental evaluation* and send packets containing signal updates to the tool. The packets can be either Boolean or continuous signal updates, or a special *termination* packet, informing the tool that the simulation is over.

STL/PSL	STL	Description
`a:x`	x	Analog variable
`phi1 - phi2`	$\phi_1 - \phi_2$	Analog operators
`phi1 + phi2`	$\phi_1 + \phi_2$	
`phi1 * phi2`	$\phi_1 * \phi_2$	
`phi1 - c`	$\phi_1 - c$	
`phi1 + c`	$\phi_1 + c$	
`phi1 * c`	$\phi_1 * c$	
`abs(phi)`	$\|\phi\|$	
`phi <= c`	$\phi \leq c$	Predicates
`phi < c`	$\phi < c$	
`phi >= c`	$\phi \geq c$	
`phi > c`	$\phi > c$	
`phi == c`	$\phi = c$	
`b:p`	p	Boolean proposition
`not varphi1`	$\neg\varphi$	Boolean operators
`varphi1 or varphi2`	$\varphi_1 \vee \varphi_2$	
`varphi1 and varphi2`	$\varphi_1 \wedge \varphi_2$	
`varphi1 -> varphi2`	$\varphi_1 \rightarrow \varphi_2$	
`varphi1 <-> varphi2`	$\varphi_1 \leftrightarrow \varphi_2$	
`varphi1 xor varphi2`	$\varphi_1 \neq \varphi_2$	
`eventually varphi`	$\diamondsuit\, \varphi$	Future temporal operators
`eventually(a:b) varphi`	$\diamondsuit_{(a,b)}\, \varphi$	
`eventually[a:b) varphi`	$\diamondsuit_{[a,b)}\, \varphi$	
`eventually(a:b] varphi`	$\diamondsuit_{(a,b]}\, \varphi$	
`eventually[a:b] varphi`	$\diamondsuit_{[a,b]}\, \varphi$	
`eventually[>b] varphi`	$\diamondsuit_{>b}\, \varphi$	
`eventually[>=b] varphi`	$\diamondsuit_{\geq b}\, \varphi$	
`always varphi`	$\square\, \varphi$	
`always(a:b) varphi`	$\square_{(a,b)}\, \varphi$	
`always[a:b) varphi`	$\square_{[a,b)}\, \varphi$	
`always(a:b] varphi`	$\square_{(a,b]}\, \varphi$	
`always[a:b] varphi`	$\square_{[a,b]}\, \varphi$	
`always[>b] varphi`	$\square_{>b}\, \varphi$	
`always[>=b] varphi`	$\square_{\geq b}\, \varphi$	
`varphi1 until varphi2`	$\varphi_1 \mathcal{U} \varphi_2$	
`varphi1 until(a:b) varphi2`	$\varphi_1 \mathcal{U}_{(a,b)} \varphi_2$	
`varphi1 until[a:b) varphi2`	$\varphi_1 \mathcal{U}_{[a,b)} \varphi_2$	
`varphi1 until(a:b] varphi2`	$\varphi_1 \mathcal{U}_{(a,b]} \varphi_2$	
`varphi1 until[a:b] varphi2`	$\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$	
`varphi1 until[>b] varphi2`	$\varphi_1 \mathcal{U}_{>b} \varphi_2$	
`varphi1 until[>=b] varphi2`	$\varphi_1 \mathcal{U}_{\geq b} \varphi_2$	

Table 6.1. STL/PSL operators

STL/PSL	STL	Description
`once varphi`	$\diamondminus\, \varphi$	Past temporal operators
`once(a:b) varphi`	$\diamondminus_{(a,b)}\, \varphi$	
`once[a:b) varphi`	$\diamondminus_{[a,b)}\, \varphi$	
`once(a:b] varphi`	$\diamondminus_{(a,b]}\, \varphi$	
`once[a:b] varphi`	$\diamondminus_{[a,b]}\, \varphi$	
`once[>b] varphi`	$\diamondminus_{>b}\, \varphi$	
`once[>=b] varphi`	$\diamondminus_{\geq b}\, \varphi$	
`historically varphi`	$\boxminus\, \varphi$	
`historically(a:b) varphi`	$\boxminus_{(a,b)}\, \varphi$	
`historically[a:b) varphi`	$\boxminus_{[a,b)}\, \varphi$	
`historically(a:b] varphi`	$\boxminus_{(a,b]}\, \varphi$	
`historically[a:b] varphi`	$\boxminus_{[a,b]}\, \varphi$	
`historically[>b] varphi`	$\boxminus_{>b}\, \varphi$	
`historically[>=b] varphi`	$\boxminus_{\geq b}\, \varphi$	
`varphi1 since varphi2`	$\varphi_1\, \mathcal{S}\, \varphi_2$	
`varphi1 since(a:b) varphi2`	$\varphi_1\, \mathcal{S}_{(a,b)} \varphi_2$	
`varphi1 until[a:b) varphi2`	$\varphi_1\, \mathcal{S}_{[a,b)} \varphi_2$	
`varphi1 until(a:b] varphi2`	$\varphi_1\, \mathcal{S}_{(a,b]} \varphi_2$	
`varphi1 until[a:b] varphi2`	$\varphi_1\, \mathcal{S}_{[a,b]} \varphi_2$	
`varphi1 until[>b] varphi2`	$\varphi_1\, \mathcal{S}_{>b} \varphi_2$	
`varphi1 until[>=b] varphi2`	$\varphi_1\, \mathcal{S}_{\geq b} \varphi_2$	
`rise(varphi)`	$\uparrow \varphi$	Events
`fall(varphi)`	$\downarrow \varphi$	
`distance(phi1,phi2,k)`	$\lvert\phi_1 - \phi_2\rvert \leq k$	Template properties
`distance(phi1,phi2,k,t,T)`	$(\lvert\phi_1 - \phi_2\rvert > k) \rightarrow \Diamond_{[0,t]}\, \Box_{[0,T-t]}\, \lvert\phi_1 - \phi_2\rvert \leq k$	
`distance(varphi1,varphi2,t,T)`	$(\neg(\varphi_1 \leftrightarrow \varphi_2)) \rightarrow \Diamond_{[0,t]}\, \Box_{[0,T-t]}(\varphi_1 \leftrightarrow \varphi_2)$	

Table 6.2. More STL/PSL operators

7

Case Studies

In this chapter we present two case studies intended to evaluate the usefulness of our property-based approach for checking the correctness of analog and mixed-signal simulation traces. The first case study is described in Section 7.1 and involves checking properties of a FLASH memory with the simulation traces provided by ST Microelectronics. The second case study is presented in Section 7.2 and involves monitoring specifications of a DDR2 memory interface component from Rambus.

The properties used in the FLASH memory case study were provided by ST Microelectonics analog designers in form of informal specifications written in English language. These properties were translated to STL/PSL matching the original requirements from the designers. This process took several iterations involving discussions on the exact meaning of the specifications. The main objective of this case study is the evaluation of the AMT tool.

The DDR2 memory interface case study concentrates rather on the specification of complex timing properties from the official specification document [Jed06] in STL/PSL. The objective is to evaluate the expressiveness of STL/PSL with respect to a realistic example used in the analog industry and identify potential weaknesses of the approach, providing useful information about new features that could be considered in the future.

7.1 FLASH Memory Case Study

The subject of the case study is the "Tricky" technology FLASH memory test chip in $0.13\mu s$ process developed in ST Microelectronics. The FLASH memory presents a good candidate for the analog case study, in that it is a digital system whose logical behavior is implemented at the analog level. Hence, it presents a direct link between the analog and the digital worlds.

For monitoring, the system under test is seen as a black box, and we do not need to know further details about the underneath chip architecture. The memory cell can be in one of the *programming*, *reading* or *erasing* modes. The correct functioning of the chip at the analog level in a given mode is determined by the behavior of a number of signals extracted during the simulation:

bl: matrix bit line terminal	**pw**: matrix p-well terminal
wl: matrix word line	**s**: matrix source terminal
vt: threshold voltage of cell	**id**: drain current of cell

The memory cell was simulated in the *programming* and the *erasing* modes for the case study, with the simulation time being 5000 *us* and 30000 *us* respectively. Four STL/PSL properties were written to describe the correct behavior of the cell in the *programming* mode and one property in the *erasing mode*. The AMT monitoring was done on a Pentium 4 HT 2.4GHz machine with 2Gb of memory. All the properties were found to be *correct* with respect to the input traces.

7.1.1 Programming Mode

The first property requires that whenever the **vt** signal crosses the threshold of 5, both **vt** and **id** have to remain continuously above 4.5 and $5 \cdot 10^{-6}$ respectively, until **id** falls below $5e - 6$ (see Figure 7.1 for the resulting signals after the evaluation).

The STL/PSL specification for this property is:

```
vprop programming1 {

  pgm1 assert:
    always (rise(a:vt>5) ->
      ((abs (a:id) > 5e-6) and (a:vt>4.5))
      until (fall(a:id>5e-6)));
}
```

The second property is split into two assertions. The first assertion **pgm1** requires that whenever the wordline **wl** is below 0.1 but will jump to above 3.8 within $15\mu s$ and the cell is not in the programming mode (translated by the absolute value of the source current **id** being below $30 \cdot 10^{-6}$), the bitline signal **bl** should cross 3.8 before the end of the simulation, and remain above that threshold continuously until the word line **wl** goes above 6, which should happen within 300 and $1500\mu s$ from the **bl** crossing. The results of the evaluation are shown in Figure 7.2.

The second assertion **pgm2** specifies that whenever the programming procedure starts (translated by the crossing of 3.8 threshold by the bitline signal **bl**), bitline should not fall below that threshold until the signal **vt** becomes higher than 5 and the absolute value of the source current **id** goes below $5 \cdot 10^{-6}$. Figure 7.3 shows the results of the **pgm2** assertion of the property.

We use the following STL/PSL specification to express the second STL/PSL property:

```
vprop programming2 {

  define b:not_pgm :=
    rise((a:wl <= 0.1) and eventually[0:15]
      (a:wl >= 3.8 and a:id >= 30e-6));
  pgm1 assert:
    always (b:not_pgm ->
      eventually (rise(a:bl>=3.8) and
         ((a:bl>=3.8) until[300:1500] (a:wl >= 6))));

  pgm2 assert:
    always (rise(a:bl >= 3.8) ->
      (not (a:bl <= 0.1) until (a:vt >=5 and
        abs(a:id) <= 5e-6)));
}
```

Erasing Mode

We first define the erasing condition that holds whenever the wordline signal **wl** is lower than -6 and p-well **pw** is above 5. The main property states that whenever an erasing condition holds, the pointwise distance between the source **s** and p-well **pw** voltages has to be smaller than 0.1 and the value of **pw** should not be greater than 0.83 from the value of bitline **bl**.

The STL/PSL specification of the property is as follows:

```
vprop erasing {
  define b:erasing_cond :=
    a:wl <= -6 and a:pw > 5;

  erasing assert:
    always (b:erasing_cond ->
      (abs (a:s-a:pw) <= 0.1)
      and (a:bl-a:pw)>-0.83));
}
```

Figure 7.4 shows some of the representative signals of the erasing property.

P-Well Driving During Programming

This property requires that whenever both bitline **bl** and wordline **wl** signals are above 2.5 threshold, the p-well signal **pw** has to be below -0.5. The evaluation results for the p-well property are shown in Figure 7.5.

The p-well property is expressed in STL/PSL using the following specification:

```
vprop pwell {
  p_well assert:
    always ((a:bl>2.5 and a:wl>2.5) ->
      a:pw<=-0.5);
}
```

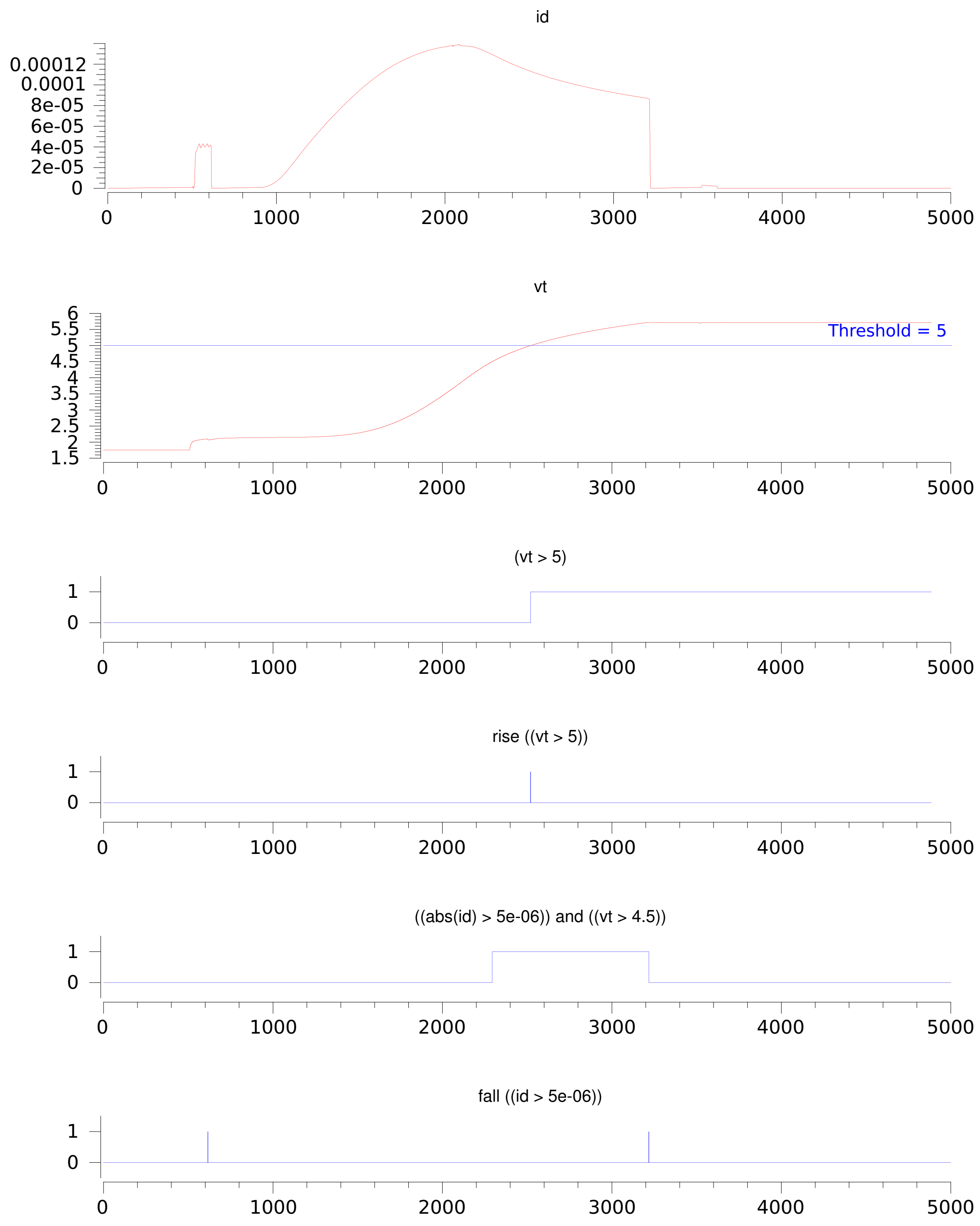


Fig. 7.1. Evaluation results for the **programming1** property

7.1.2 Tool Performance

The time and space requirements of AMT were studied with both *offline* and *incremental* algorithms. The complexity of the algorithm used in AMT is shown to be $O(k \cdot m)$

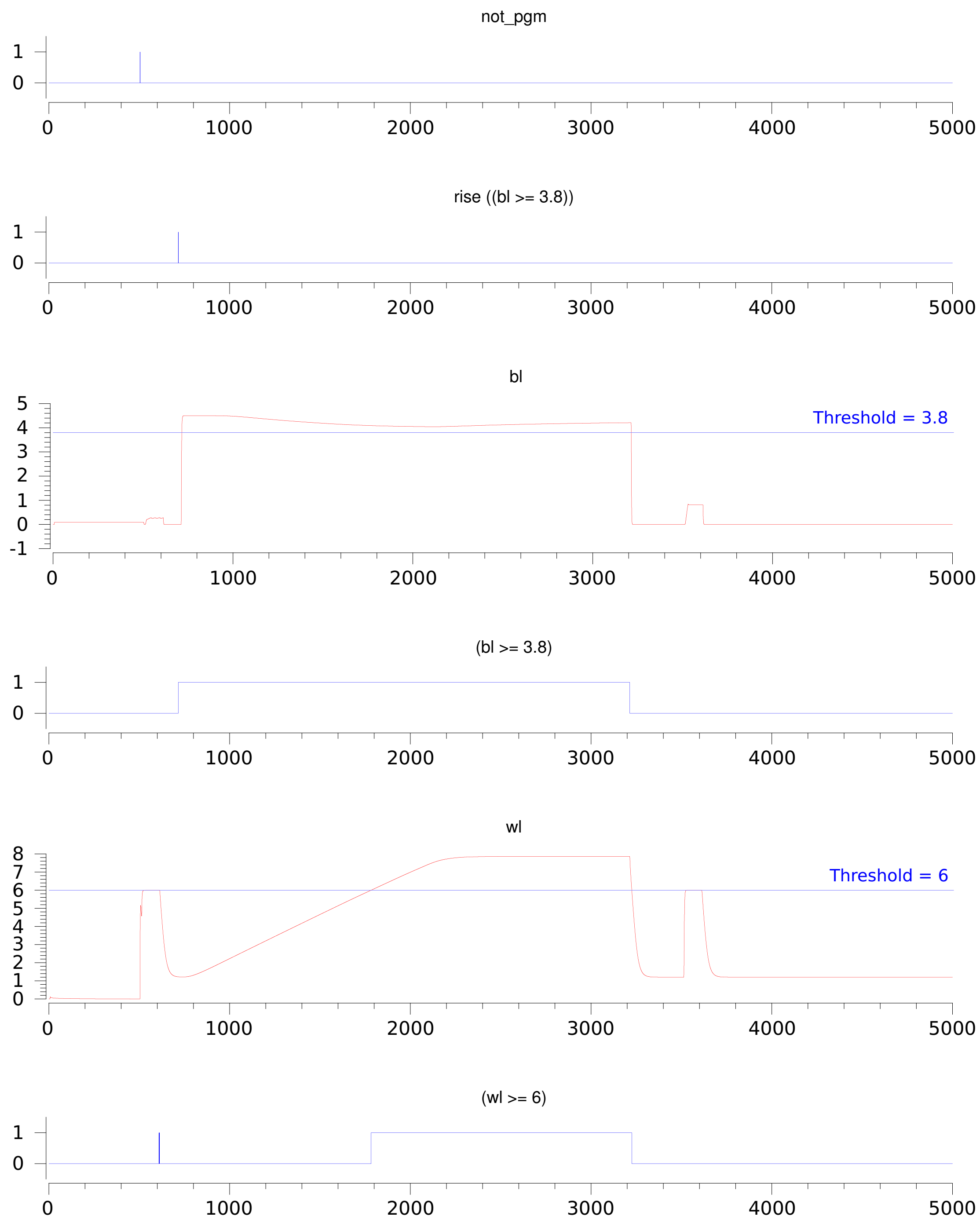


Fig. 7.2. Evaluation results for the **programming2** property (assertion **pgm1**)

in [MN04] where k is the number of sub-formulae and m is the size of the input signal (number of singular points and open segments).

Table 7.1 shows the size of the input signals (number of singular points and segments). We can see that the *erasing* mode simulation generated 10 times larger inputs from the *programming* mode simulation. Table 7.2 shows the evaluation results for the

name	pgm sim input size	erase sim input size
wl	34829	283624
pw	25478	283037
s	33433	282507
bl	32471	139511
id	375	n/a

Table 7.1. Input Size

property	time (s)	size
programming1	0.14	99715
programming2	0.42	405907
p-well	0.12	89071
erasing	2.35	2968578

Table 7.2. Offline algorithm evaluation

offline procedure of the tool. Monitoring the properties for the programming mode required less than half a second. Only the `erasing` property took more than 2 seconds, as it was tested against a larger simulation trace. We can also see that the evaluation time is linear in the size of signals generated by the procedure and can deduce that the procedure evaluates about 1,000,000 intervals per second.

Property	Offline t = total size	Incremental m = max active size	m/t * 100
programming1	99715	65700	65.9
programming2	594709	242528	40.8
p-well	89071	8	0.01

Table 7.3. Offline/incremental space requirement comparison

The execution times of the incremental algorithm are less meaningful because the procedure works in parallel with the simulator and the evaluation time depends on the frequency of the incoming input. In fact, a major advantage of the incremental procedure is the ability to detect property violation in the middle of the simulation and save simulation time. Another advantage of the incremental algorithm is its reduced space requirement as we can discard parts of the simulation after they have been fully used. Table 7.3 compares the memory consumptions of the offline and incremental procedures. For the former we take the total number of signal segments generated by the tool while for the latter we take the maximal number of signal segments kept simultaneously in memory. We can see that this ratio varies a lot from one property to another, going from 0.01% up to almost 70%. The general observation is that pointwise operators require considerably less memory in the incremental mode, while properties involving the nesting of untimed temporal properties often fail to discard their inputs until the end of the simulation.

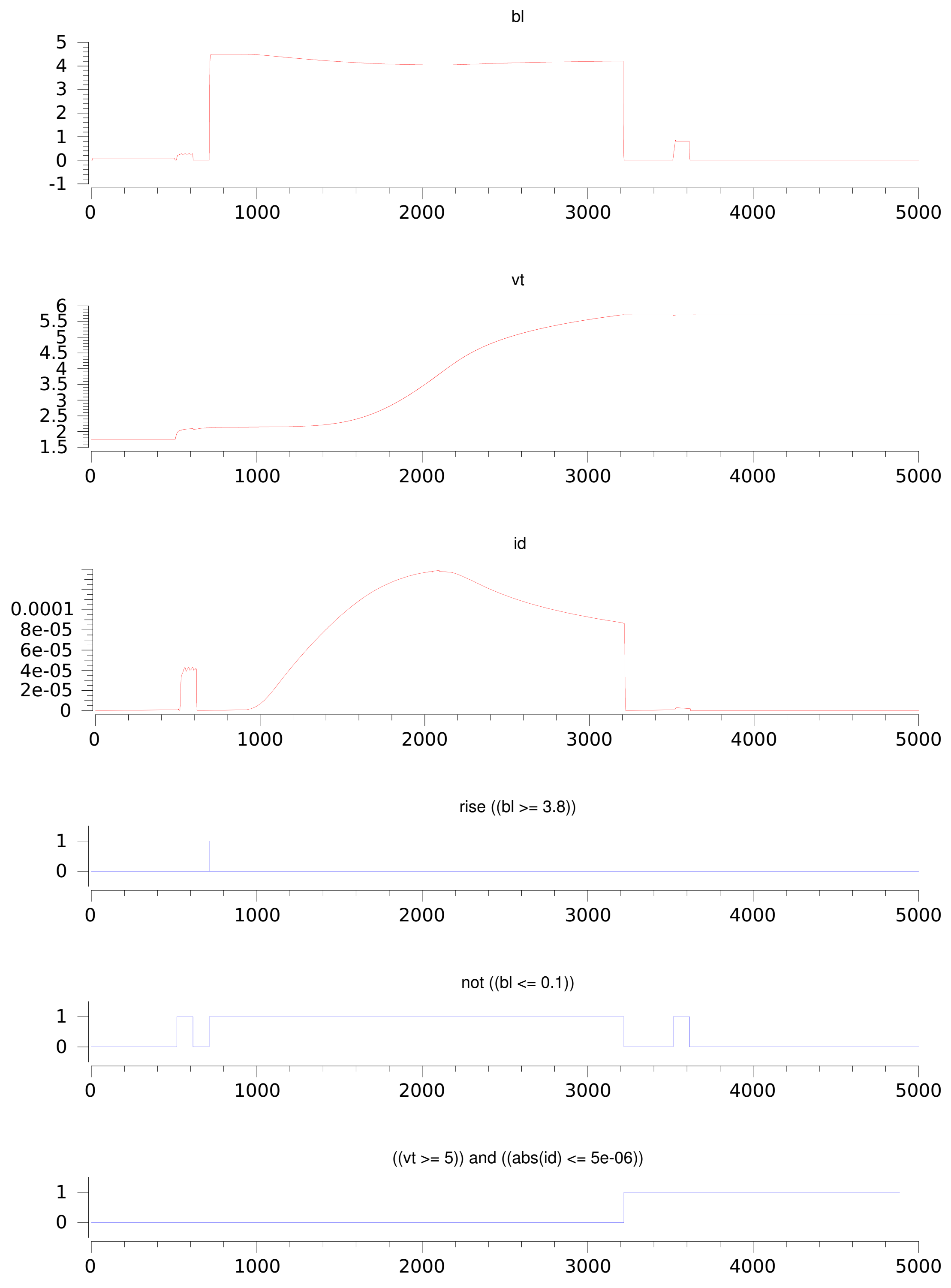


Fig. 7.3. Evaluation results for the **programming2** property (assertion **pgm2**)

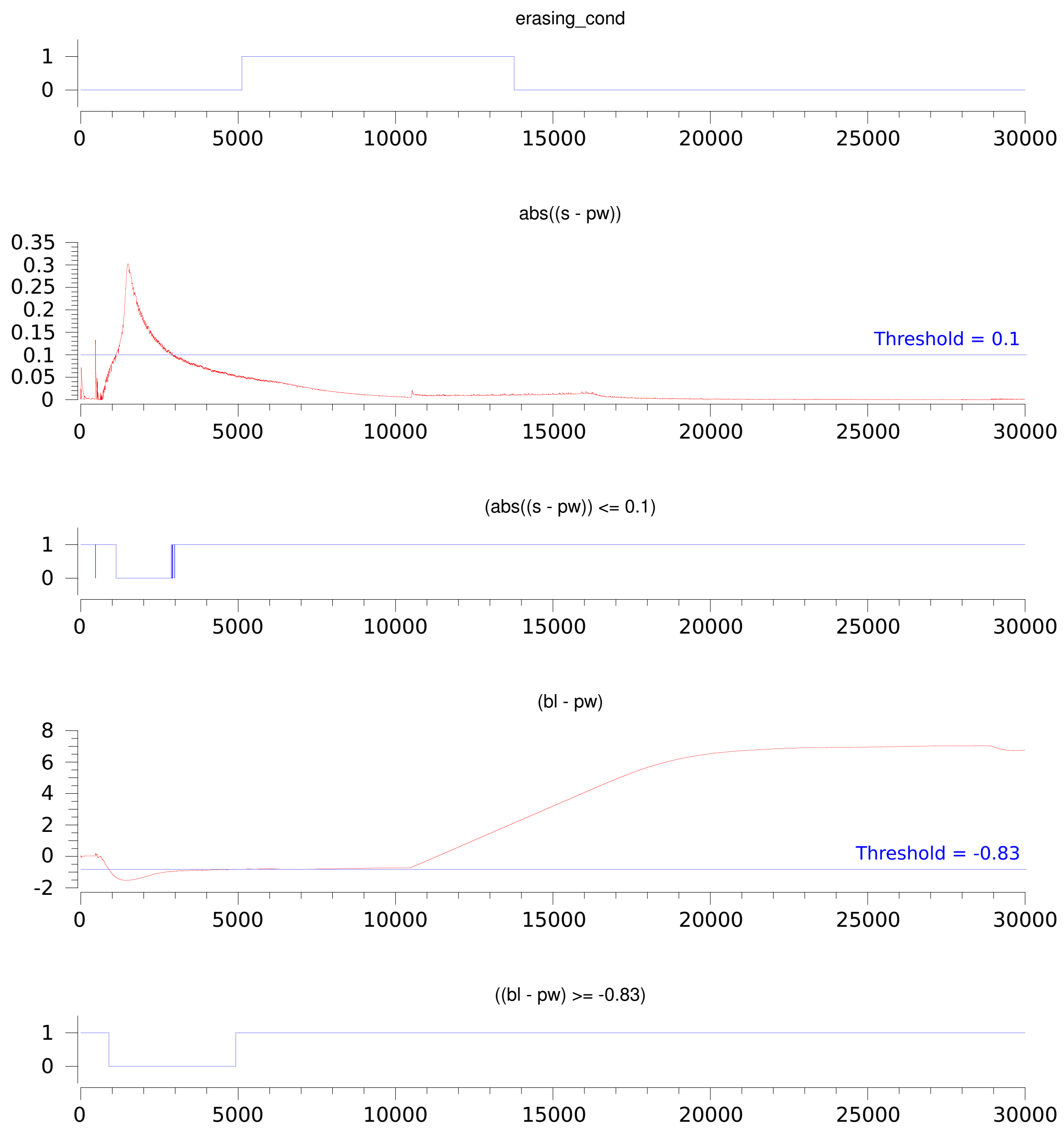


Fig. 7.4. Evaluation results for the **erasing** property

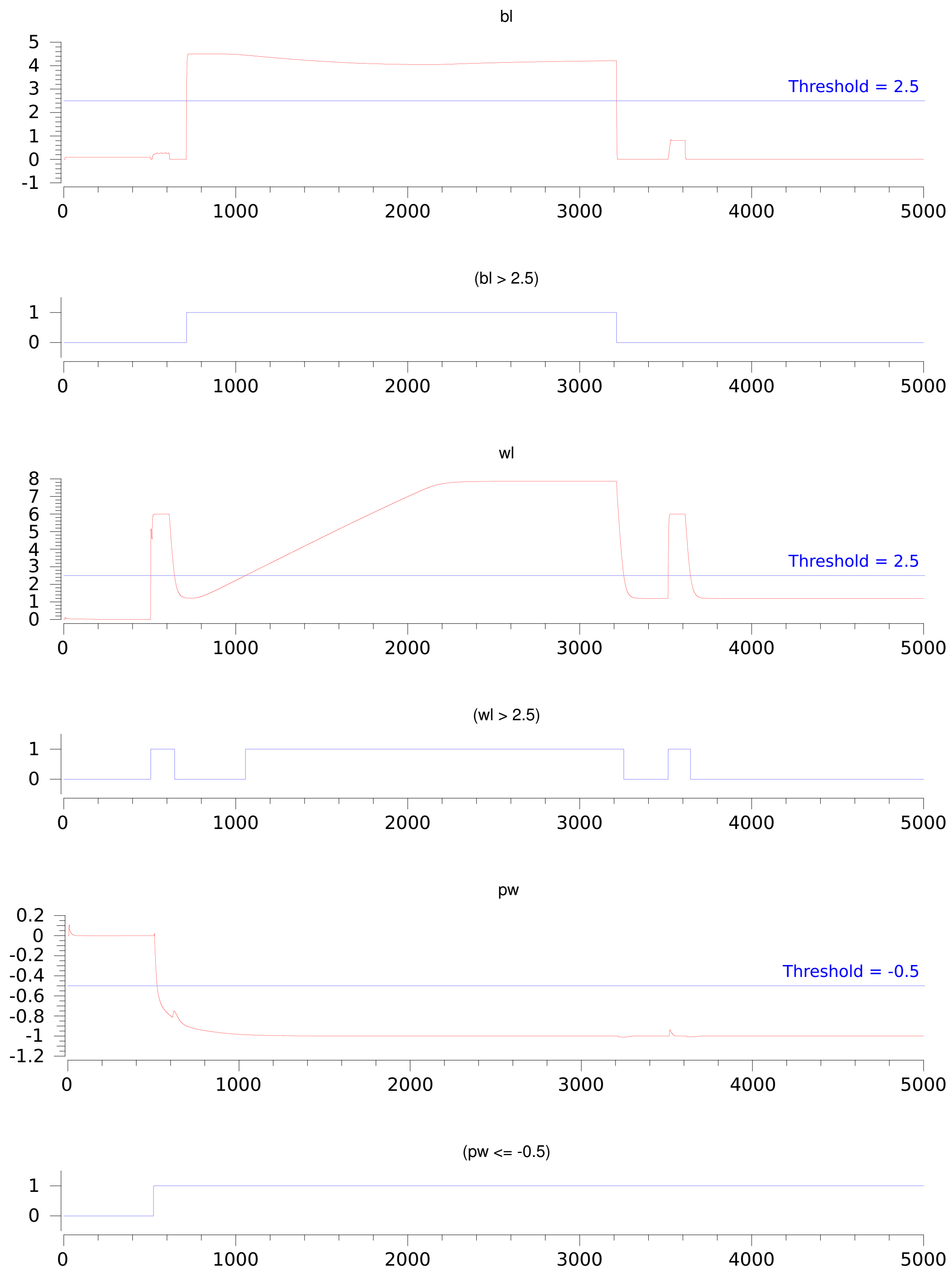


Fig. 7.5. Evaluation results for the **pwell** property

7.2 DDR2 Case Study

The subject of this case study is a DDR2 memory interface developed at Rambus. DDR2 presents a number of features that make it a good candidate for property-based monitoring approach. The memory interface acts as a bus between the memory and other components in the circuit and exhibits the communication of digital data implemented at the analog level. Hence, the correct functioning of a DDR2 memory interface largely depends on the appropriate timing of different signals within the circuit. In section 7.2.1, we describe an alignment property as a typical DDR2 property and different steps needed for translating it in an STL/PSL specification. The experimental results are presented in 7.2.3.

7.2.1 Alignment Between Data and Data Strobe Signals

In DDR2, the data access is controlled by a single-ended or differential data strobe signal, which acts as an asynchronous clock. The official JEDEC DDR2 specification is defined in [Jed06] and describes, amongst others, a number of properties that involve timing relationship between events that happen in the data and data strobe signals. In this case study, we are particularly interested in a property that defines the correct alignment between these two signals. The case study considers the specification parameters for the single-ended data strobe DDR2-400 memory interface, which is part of the JEDEC standard.

The DDR2 specification contains a number of relevant thresholds, shown in Table 7.4. The temporal relationship between data signal DQ and data strobe signal DQS is defined with respect to the crossings of these thresholds.

Threshold	Value (V)
V_{DDQ}	1.8
$V_{IH(AC)_{min}}$	1.25
$V_{IH(DC)_{min}}$	1.025
$V_{REF(DC)}$	0.9
$V_{IL(DC)_{max}}$	0.775
$V_{IL(AC)_{max}}$	0.65

Table 7.4. Threshold values for DQ and DQS

The general definition of the alignment of data DQ and data strobe DQS signals is shown in Figure 7.6. The proper alignment between the two signals is determined by two values, the *setup* time tDS and *hold* time tDH. The setup and hold times of DQ and DQS are checked both on their *falling* and *rising* edges, but we only consider, for the sake of simplicity, the specification of the property for the setup time at the falling edge of the signals (the other cases are similar and symmetric).

Informally, the setup property at the falling edge requires that whenever DQS crosses the $V_{IH(DC)_{min}}$ threshold from above, the previous crossing of $V_{IL(AC)_{max}}$ by the signal

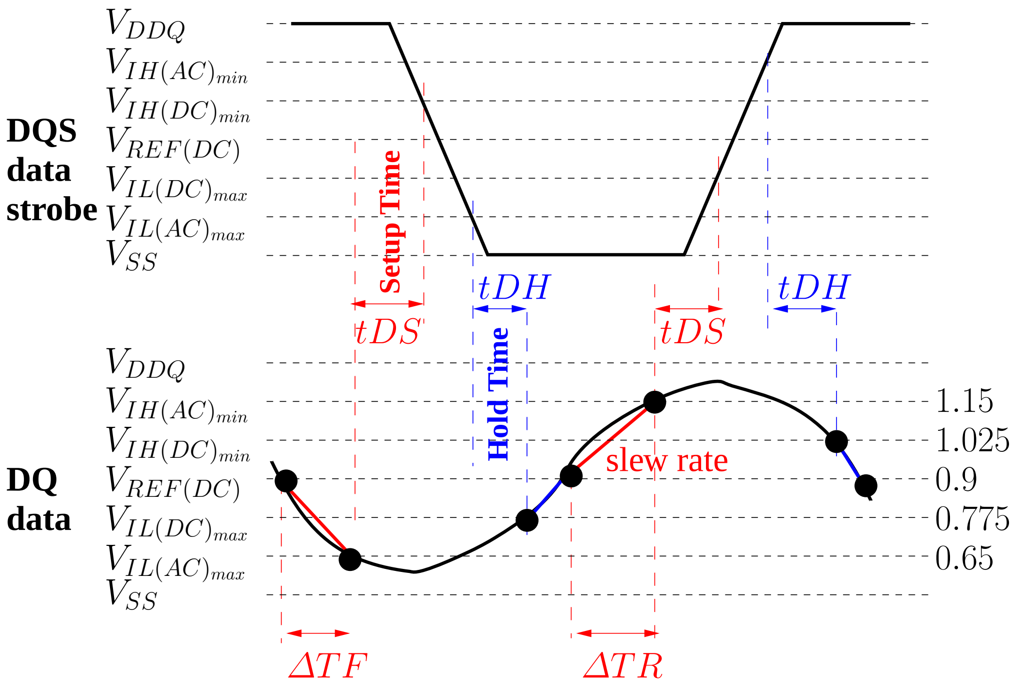


Fig. 7.6. Data DQ and data strobe DQS alignment

DQ from above should precede it by at least a period of time of tDS. This property is formalized in STL/PSL as follows

```
define b:dqs_above_vihdcmin := (a:DQS >= 1.025);
define b:dqs_above_vilacmax := (a:DQ >= 0.65);

always (fall(b:dqs_above_vihdcmin)
  -> historically[0:tDS] not fall(b:dq_above_vilacmax));
```

Unfortunately the above property, naturally expressed in STL/PSL, does not present the full reality. In fact, setup time tDS is not a constant value, but rather varies according to the slew rates (slopes) of DQ and DQS signals. For example, when DQ and DQS fall more sharply, the required tDS increases. Setup time tDS is equal to the sum of a (constant) *base term* $tDS(base)$ and a (variable) *correction term* ΔtDS

$$tDS = tDS(base) + \Delta tDS$$

The setup base term $tDS(base)$ is equal to $150ps$ for the single-ended DDR2-400. The correction term ΔtDS is a value that depends directly on slew rates of DQ and DQS, with the setup slew rate of a falling signal being defined as

$$sr = \frac{V_{REF(DC)} - V_{IL(AC)_{max}}}{\Delta TF} \tag{7.1}$$

where ΔTF is the time that the signal spends between $V_{REF(DC)}$ and $V_{IL(AC)_{max}}$. As we can see, the falling setup slew rate sr of a signal can be deduced from ΔTF.

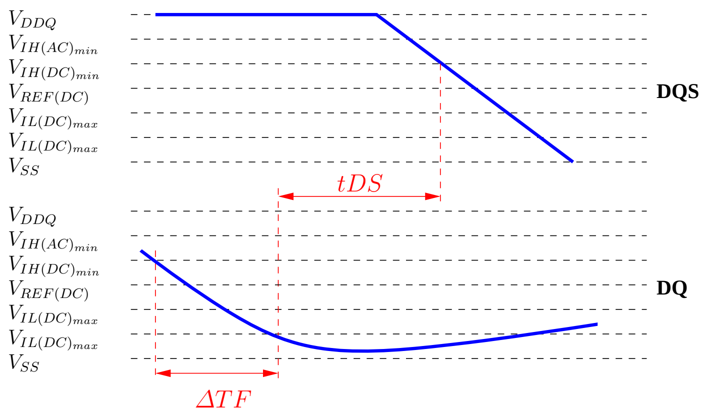


Fig. 7.7. DQ/DQS falling setup time tDS and the relation between slew rate and ΔTF

In order to extract the setup correction term ΔtDS from the actual slew rates of DQ and DQS (sr_{DQ} and sr_{DQS}), we can use a specification table from [Jed06], partially reproduced in Table 7.5. According to the JEDEC specification, ΔtDS corresponding to the slew rates not listed in Table 7.5 should be linearly interpolated. Consequently, we can apply the following sequence of computations in order to determine the correct value of tDS at any time

$$\Delta TF \longrightarrow \text{setup falling slew rate} \longrightarrow \text{correction term} \longrightarrow tDS$$

		DQS Single-Ended Slew Rate tDS			
		2V/ns	1.5V/ns	1V/ns	0.9V/ns
DQ	2V/ns	**188**	167	125	
Single-	1.5V/ns	146	125	83	81
Ended	1V/ns	63	42	0	-2
Slew Rate	0.9V/ns		31	-11	13
tDS					

Table 7.5. Correction terms for setup time

To summarize, tDS is a value that varies during the simulation as a function of slew rates of DQ and DQS ($tDS = f(sr_{DQ}, sr_{DQS})$). The problem is that STL/PSL cannot capture parameterized time bounds and therefore we have to use approximation in order to express a similar alignment property that still preserves some guarantees. We can subdivide the domain of slew rates (say $R = [sr_{min}, sr_{max}]$) into n regions $R_1, \ldots, R_n$. For each pair (R_i, R_j) of DQ/DQS slew rate regions, we assign a separate constant setup time tDS_{ij}. Instead of one property, we will have $n \times n$ properties of the form: "whenever DQS crosses the $V_{IH(DC)_{min}}$ threshold from above, DQ slew rate sr_{DQ} is in R_i and DQS slew rate sr_{DQS} is in R_j, the previous crossing of $V_{IL(AC)_{max}}$ by the signal DQ from above should precede it by at least a period of time of tDS_{ij}."

The proper constant value for tDS_{ij} for a pair of slew rate regions (R_i, R_j) can be chosen in two different manners. The first solution consists in computing tDS_{ij} from the maximum correction term for the DQ and DQS slew rates that are in the R_i and R_j regions, respectively. This corresponds to an over-approximation of the original specification, and if this property is violated, we don't know if it is a real failure or a false alarm. On the other hand, the satisfaction of the over-approximated property implies that the original one holds too. Conversely, the computation of tDS_{ij} from the minimum correction term defined for the slew rates in the pair of regions (R_i, R_j) yields to an under-approximation of the original property. If the new property is falsified, we know that it corresponds to a real violation, while if it passes, we cannot say whether we are indeed safe.

As an example, consider the highlighted range of Table 7.5, which we call the "top-left" range, where the setup falling slew rates of DQ and DQS are between 1 and $2\ V/ns$. For the conservative approximation of tDS, with slew rates falling in that range, we choose the worst-case ΔtDS as the correction term, that is $188ps$. Hence, the approximated falling setup time tDS_{TL} for all DQ and DQS with falling slew rates between 1 and $2V/ns$ would be equal to $tDS_{TL} = 150 + 188 = 338ps$.

In order to determine the falling slew rates of DQ and DQS, we need to detect how much time these signals remain in their falling slew region (between $V_{REF(DC)}$ and $V_{IL(AC)_{max}}$crossing $V_{REF(DC)}$ from above). This can be done with the following formula

```
define b:dq_in_fsr :=
  ((a:DQ <= 0.9) and (a:DQ >= 0.65))
    since (a:DQ >= 0.9);

define b:dqs_in_fsr :=
  ((a:DQS <= 0.9) and (a:DQS >= 0.65))
    since (a:DQS >= 0.9)
```

which holds if the signal is in the falling slew region, as shown in Figure 7.8.

Note that according to equation (1), DQ and DQS have their slew rates in the range between 1 and $2V/ns$ if their respective ΔTF is between 125 and $250ps$. Moreover, the value of tDS is determined at the crossing of $V_{REF(DC)}$ by DQS from above (point **ref** in Figure 7.9) with respect to the previous falling setup slew rate of DQ and the next falling setup slew rate of DQS, as shown in Figure 7.9. Hence, the falling slew rates of DQ and DQS are in the range between 1 and $2V/ns$ if the following formulae hold

```
define b:dq_slew_rate_in_1_2 :=
  not b:dq_in_fsr since
   (b:dq_in_fsr since[125:250) (rise(b:dq_in_fsr)));

define b:dqs_slew_rate_in_1_2 :=
  not b:dqs_in_fsr until
    (b:dqs_in_fsr until[125:250) (fall(b:dqs_in_fsr)));
```

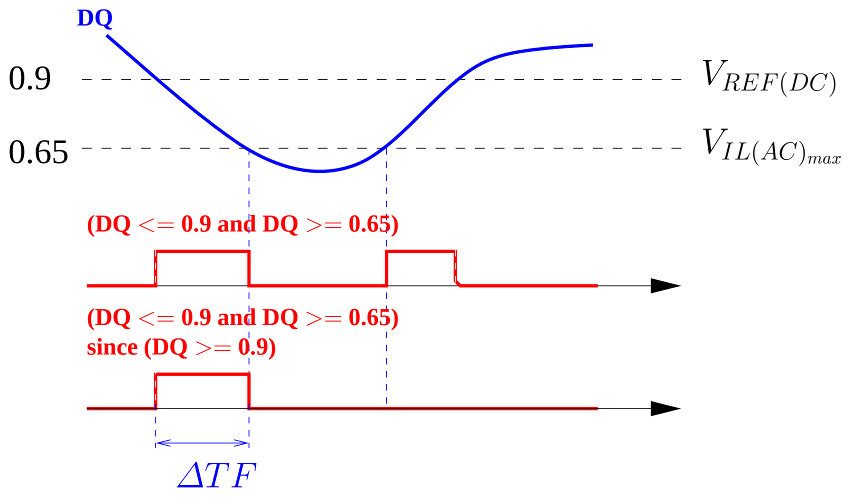


Fig. 7.8. Falling slew region and ΔTF

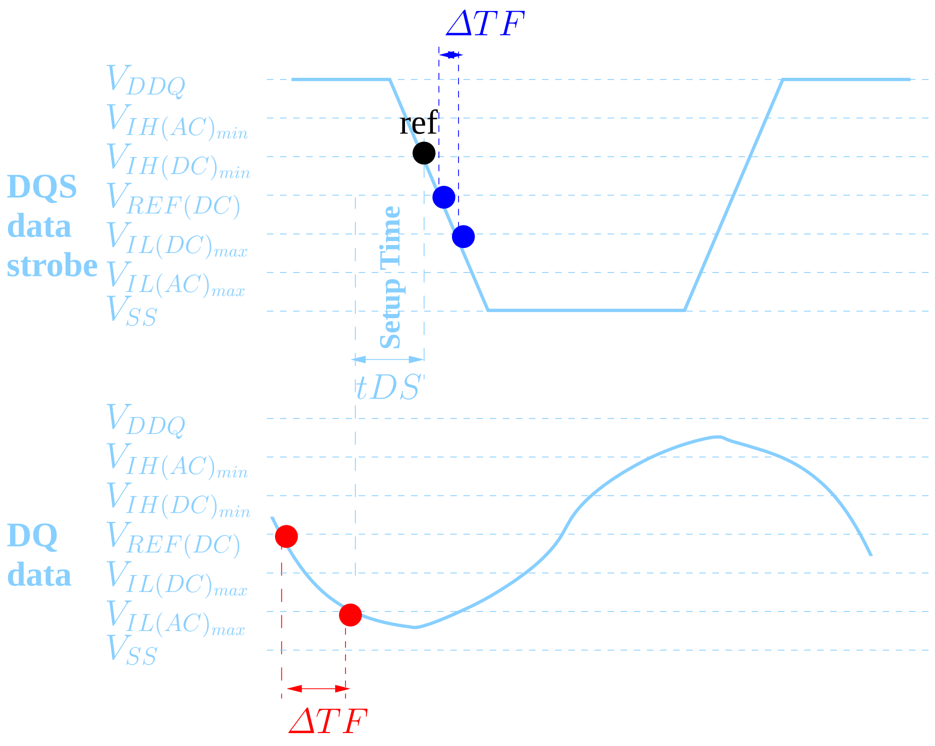


Fig. 7.9. Relation between the reference point and the corresponding ΔTF of DQ and DQS

```
define b:top_left_region :=
    b:dq_slew_rate_in_1_2 and
      b:dqs_slew_rate_in_1_2;
```

Finally, the main property for the falling setup time, provided that DQ and DQS falling slew rates are in the range between 1 and $2V/ns$, is expressed as

```
define b:dqs_above_vihdcmin := (a:DQS >= 1.025);
define b:dqs_above_vilacmax := (a:DQ >= 0.65);

always ((fall(b:dqs_above_vihdcmin)
  and b:top_left_region)
  -> historically[0:338] not fall(b:dq_above_vilacmax));
```

with similar properties that have to be written for each range of DQ and DQS slew rates.

7.2.2 Methodological Evaluation

Property-based monitoring of analog and mixed-signal behaviors is a novel approach and it is worth discussing some methodological aspects related to this case study. The process started by investigating the validation methods that are currently used by analog designers and understanding what are the actual difficulties that they encounter in checking the correctness of their designs. The next step required to identify the type of application whose validation is not fully covered by existing tools and that could benefit from assertion-based monitoring techniques, which led us to consider the DDR2 memory interface. With the help of analog designers we were able to study in detail different properties that are defined in the official DDR2 specification, and consequently understand how to translate them into STL/PSL assertions. This preparation process of the case study is difficult to quantify although it clearly took orders of magnitude more time than the actual writing and evaluation of the assertions that describe DDR2 properties. Despite the length of this pre-processing, it was a crucial step in understanding relevance, strengths and weaknesses of the property-based analog monitoring framework.

7.2.3 Experimental Evaluation

In this case study, we considered a single-ended DDR2-1066 memory interface, which is not yet a JEDEC standard. Hence the exact specification parameters could not be obtained for that particular version of DDR2, and we used instead the official specification parameters for the single-ended DDR2-400 presented in Section 7.2.1, assuming that these parameters would be conservative enough. The simulation traces contained about 180,000 samples for each signal. We used the offline monitoring for this case study because the DDR2 simulation traces were already available.

The translation of the alignment property into a set of STL/PSL assertions started by splitting the main property into 4 different ranges, taking an over-approximated tDS value for each slew rate range. The evaluation of each property took about 7 seconds. Since some of the over-approximating properties were shown to be false, we decomposed them further in 3 iterations into a total of 7 properties before being able to show that the simulation traces satisfy the specification. The properties were refined manually and this proved to be a tedious task.

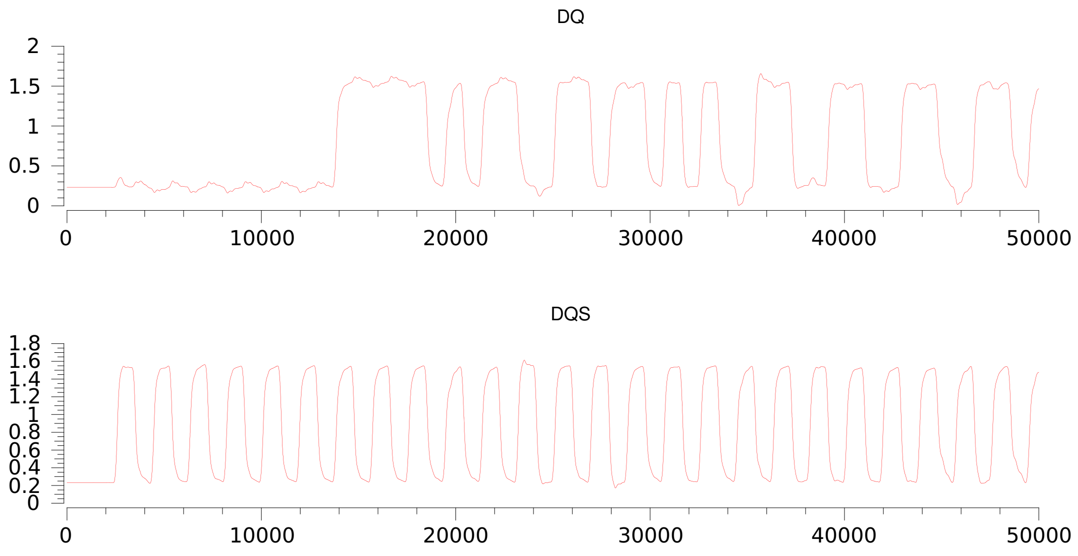


Fig. 7.10. Segment of DQ and DQS simulation traces

7.3 Conclusions

The FLASH and DDR2 case studies present, to the best of our knowledge, first attempts to apply property-based monitoring framework to a realistic analog industrial designs. The importance of these case studies lies in the fact that they exposed the relevance and the level of maturity of assertion-based methodology in the context of analog validation.

The case studies showed that an important class of non-trivial properties describe event-based timing relationships between analog signals, which can be in general naturally expressed in a specification language such as STL/PSL. Since assertion checking remains a "lightweight" simulation-based validation technique, it fits well with the current practice of analog designers. We believe that this methodology can provide an extra set of useful checks on simulation traces, which are already generated by the designers for their own purposes. Moreover, in the analog domain it often takes orders of magnitude longer to produce simulation traces than to check assertions. Consequently, the overhead induced by property monitors with respect to simulation time remains low, while it can provide another level of confidence in the correct functioning of the underlying design. In our opinion, the general idea of simulation-based checking of properties to find potential bugs may be successfully adapted from digital to analog and mixed-signal domain and integrated into the analog validation flow in a reasonably-near future.

The DDR2 case study also revealed some weaknesses in the current state of analog property checking, providing useful guidelines for further development and optimization of this methodology. For instance, the timing relationship between analog signals can be more complex than what STL/PSL (and MITL) can express. This problem has been exposed by the DDR2 data vs. data strobe alignment property. We had to use approximate techniques in order to show that the alignment between data and data strobe signals was correct. Consequently, the resulting specification turned out to be quite complex to write. Another difficulty is related to the fact that STL/PSL is based on a temporal logic, a formalism that remains esoteric to analog designers[1]. Consequently, we should consider identifying some common properties encountered by analog designers, and use parameterized templates to "hide out" the temporal logic details.

We present here some directions for future work based on different observations made during the evaluation of the case studies:

1. **Parameterized time bounds:** the DDR2 case study showed that STL/PSL temporal operators with constant time bounds may not be sufficient to describe some realistic relations between analog signals. The temporal relations between events in input signals require more flexibility, such as time bounds that are functions of parameters that vary during the simulation.
2. **Tighter integration with simulators:** property-based analog checking approach would be more appealing to designers if the specification and monitoring process were embedded in the standard design languages and simulators. In the digital world, the assertions are often integrated into Verilog or VHDL code and are inserted at the

[1] It might be the case that the verification task will be carried out by digital designers at the system integration phase, which will make the "cultural" problems less severe. However, this observation opens the question of what properties are most beneficial to integration within the property-based monitoring approach.

points where the property should be checked. A tighter integration of analog and mixed signal assertions into the current design flow would consist of the following steps:

a) Standardization of the language, a step that could convince EDA companies to consider integrating assertion-based AMS validation methodology into their tools, and would encourage designers to use such assertions in their designs. STL/PSL follows this direction as it extends the existing standard PSL constructs. Due to the importance of the SVA specification language in the digital domain, we would also need to consider analog and mixed-signal extensions of SVA.
b) Integration of assertions into VERILOG-AMS and VHDL-AMS code. Designers prefer inserting assertions at the points in their design which they want to check, than having a separate tool rather used solely for specification and evaluation of the properties. This tight integration would bring other benefits, such as the possibility to use existing VERILOG/VHDL-AMS constructs within the assertions (better detection of threshold crossing using `@cross`, express richer properties using derivatives and integrals, etc.). Finally, property monitors would be embedded into the simulation process, and could stop it when an assertion is violated and hence save simulation time.

3. **Property-based parameter extraction:** the interaction with analog designers revealed that the verification with respect to the existing specification is not the only interesting question that can be asked about an analog design. In fact, the specification parameters such as timing relationship between different signals are often not known in advance. Such parameters are rather *extracted* from the simulation traces, and the specification is completed only after simulating a model of the design. We would like to express properties without specifying the time bounds, for example `always (rise(b:p) -> eventually![?] b:q)`, asking the following question: given a set of simulation traces, what are the minimum and maximum time bounds, if any, such that the the property is satisfied. In formal methods community, this problem is known as model measuring, and has been considered in the context of parametric temporal logics in [AELP99].
4. **Integration with test generation:** an interesting direction of research would be to combine the property-based AMS checking approach with techniques for automatic generation of simulation traces, such as those studied in [ND07a, ND07b]. Such a combination could make the analog validation process more automatic.
5. **More comprehensive examples:** the case studies carried out in this thesis pointed out the classes of analog properties that are natural to express in a specification language like STL/PSL, but more importantly helped us to identify possible extensions of the language that would increase its expressiveness and make the specification process easier to the analog designer. Applying the property-based validation methodology to other industrial analog and mixed-signal design examples would provide additional useful information about the robustness of this approach and guide our future work on extending the specification language.

8

From MITL to Timed Automata

In this section we show how to build for every MITL formula φ a temporal tester, a timed signal transducer which computes the characteristic function of φ. We assume that that the formula has been rewritten to a form which uses only the 6 operators of Proposition 3.3.

8.1 Temporal Testers for $p\mathcal{S}q$ and $p\mathcal{U}q$

Proposition 8.1. *One can construct a temporal tester that realizes* $\chi^{p\mathcal{S}q}$.

The construction of the tester for $p\mathcal{S}q$ is similar to the untimed case. The tester reads the input signal w and decides at every time instant t the output value u depending on the history of the observed values of p and q. We have shown in Lemma 3.4 that $p\mathcal{S}q$ is left-continuous, meaning that the satisfaction of the operator at any singular point of the signal cannot differ from its satisfaction during the preceding open segment. In Lemma 3.6, we provided a number of rules that determine the value of u in an open segment based on the values of p and q in that segment and the preceding singular point, summarized in Table 8.1. The combination of these two results gives direct guidelines for constructing the tester for $p\mathcal{S}q$.

Case	$\dot{w}_i$	w_i	u_i
1	$*$	$\overline{p}$	0
2	$*$	pq	1
3a	$\overline{pq}$		0
3b	q	$p\overline{q}$	1
3c	$p\overline{q}$		$\dot{u}_i$

Table 8.1. Rules of Lemma 3.6 relating u_i with w_i and $\dot{w}_i$ for $u = \chi^{p\mathcal{S}q}(w)$

The temporal tester for $p\mathcal{S}q$ is shown in Figure 8.1 and is constructed as follows. Following Lemma 3.4, the output at time 0 is $\overline{u}$ irrespective of the initial input values. Lemma 3.4 also requires that the output at any singular point has to agree with the output

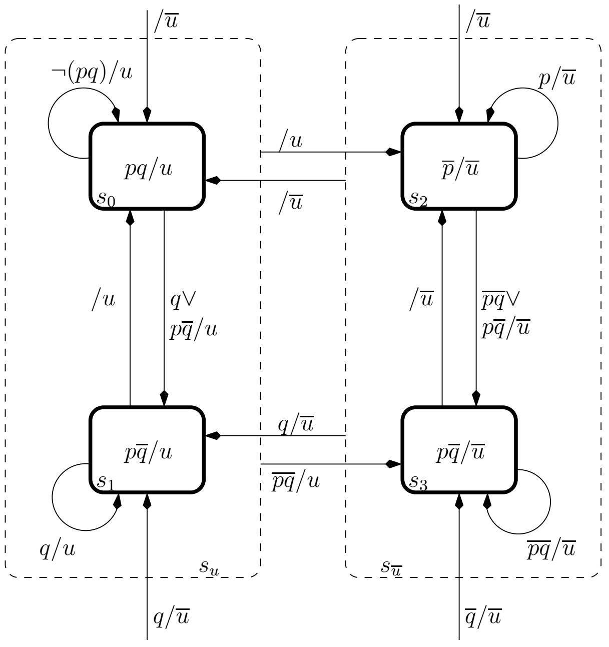


Fig. 8.1. Temporal tester for $p\mathcal{S}q$. The states are grouped, according to their outputs into $s_u = \{s_0, s_1\}$ and $s_{\overline{u}} = \{s_2, s_3\}$.

during the preceding open segment. This fact is realized in the tester by having the output labels on the transitions agreeing with the labels in the source locations. During an open segment $(w_i)_{r_i}$ of w, the tester reads the inputs and generates the corresponding output segments $(u_i)_{r_i}$ according to the rules in Lemma 3.6, which relate the output value u_i the input values w_i and $\dot{w}_i$. When an open segment $\overline{p}$ is read, the tester is in location s_2 and outputs $\overline{u}$ (case 1). The output value does not depend on the inputs at the previous singular point, hence the ingoing transitions can have any input label. The only exception is the self-loop $s_2 \longrightarrow s_2$ which is labeled by p in order to avoid a transition labeled by $\overline{p}$ taking place at a singular point of the signal. Similarly, in location s_0, the segment labeled by pq is read and the output is u (case 2) and the incoming transitions can have any input label (again, except for the self-looping transition). When considering case 3, where a $p\overline{q}$ segment is read, the situation is more involved, since the output value can be either u (location s_1) or $\overline{u}$ (location s_3) depending on the values of the input preceding the $p\overline{q}$ open segment. In the case that the singular point preceding the $p\overline{q}$ segment was labeled by $\overline{pq}$ (case 3-(a)), the output during the segment is $\overline{u}$ (transitions $s_2 \longrightarrow s_3$, $s_3 \longrightarrow s_3$ and $s_u \longrightarrow s_3$). If q was true at the singular point prior to the $p\overline{q}$ segment (case 3-(b)), the tester outputs u (transitions $s_0 \longrightarrow s_1$, $s_1 \longrightarrow s_1$ and $s_{\overline{u}} \longrightarrow s_1$). Finally, if the value read at the singular point preceding the $p\overline{q}$ segment was also $p\overline{q}$ (case 3-(c)), the output value has to agree with the output value at that singular point (transitions $s_0 \longrightarrow s_1$ and $s_2 \longrightarrow s_3$). In other words there are two $p\overline{q}$ states that differ in their history. Location s_1 is entered via histories that make $p\mathcal{S}q$ satisfied while s_3 is entered via histories that falsify it. From this follows that all runs of the automaton satisfy Lemma 3.4 and Lemma 3.6 and thus are consistent with the semantics of *since*. Observing that the automaton is non-blocking and every input has an infinite run, we can conclude the proof of Proposition 8.1. $\square$

Proposition 8.2. *One can construct a temporal tester that realizes* $\chi^{p\mathcal{U}q}$.

The tester for $p\mathcal{U}q$ is similar to the one of $p\mathcal{S}q$. We have shown in Lemma 3.5 that $p\mathcal{U}q$ is right-continuous, meaning that it is satisfied at some singular point t iff it is also satisfied in its right neighborhood. In Lemma 3.7 we provided a set of rules, summarized in Table 8.1 that relate the value of u in an open segment to the values of p and q in that segment and the subsequent singular point. The combination of these two results provides rules for the tester construction.

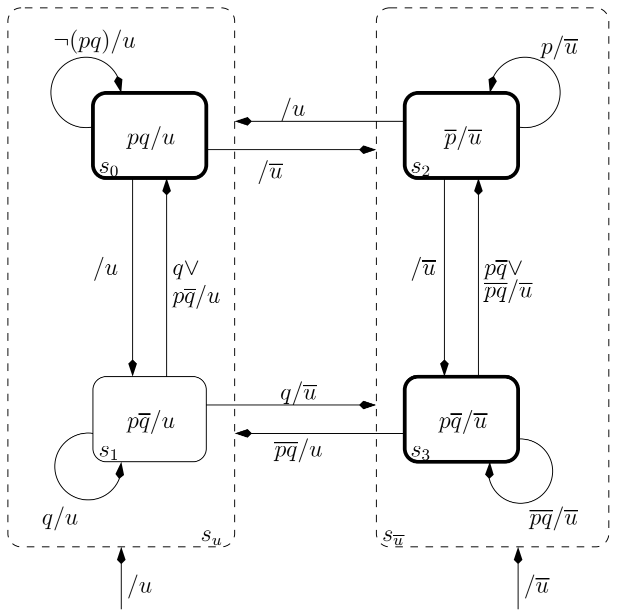


Fig. 8.2. Temporal tester for $p\mathcal{U}q$

Case	u_i	w_i	$\dot{w}_{i+1}$
1	0	$\overline{p}$	$*$
2	1	pq	$*$
3a	0		$\overline{pq}$
3b	1	$p\overline{q}$	q
3c	$\dot{u}_{i+1}$		$p\overline{q}$

Table 8.2. Rules of Lemma 3.7 relating u_i with w_i and $\dot{w}_{i+1}$ for $u = \chi^{p\mathcal{U}q}(w)$

The temporal tester for $p\mathcal{U}q$, shown in Figure 8.2, is symmetric to the one of $p\mathcal{S}q$ and is obtained from it by simply inverting the transition arrows. Unlike its past counterpart which reads inputs, and determines the output according to the observed history, the tester for $p\mathcal{U}q$ predicts the output non-deterministically, and the predictions have to be confirmed by future inputs (wrong predictions are aborted). Lemma 3.5 requires that at all singular points, the prediction of the output value has to agree with the output value in the next open segment. This fact is realized in the tester by letting output labels on transitions be identical to the labels at the target location. During open segments of w, the tester generates outputs which have to be confirmed by future inputs according to the rules of Lemma 3.7. When an open $\overline{p}$ segment is observed, the tester is at location s_2 and the output is $\overline{u}$ (case 1). The prediction is immediately confirmed and is independent of the input value at the subsequent singular point. Hence, the outgoing transitions, except for the self-loop, can have any input label. Similarly, when an open pq segment is observed (location s_0), the output is u and the outgoing transitions, except for the self-loop, can have any value. Finally, when an open $p\overline{q}$ segment is read, the situation is more involved, as the tester can make two different predictions non-deterministically, generating two separate runs, one of which will be aborted later. This situation is realized by having two separate locations s_1 and s_3, both labeled by $p\overline{q}$, and predicting the outputs u and $\overline{u}$, respectively. After observing a $p\overline{q}$ segment there are three possibilities: 1) the prediction $\overline{u}$ (location s_3) is followed by a singular point labeled by $\overline{pq}$ (case 3-(a) of Lemma 3.7)). This situation is realized in the tester by location s_3 having outgoing transitions labeled by $\overline{pq}$ (transitions $s_3 \rightarrow s_3$, $s_3 \rightarrow s_2$ and $s_3 \rightarrow s_u$); 2) Similarly, the tester is at location s_1 predicting u and the segment $p\overline{q}$ is followed by a q-labeled singular point (see case 3-(b) of Lemma 3.7) and the tester takes one of the q-labeled transitions $s_1 \rightarrow s_1$, $s_1 \rightarrow s_0$ or $s_1 \rightarrow s_{\overline{u}}$; 3) finally, the open segment $p\overline{q}$ is followed by a singular point labeled by $p\overline{q}$. In this case, neither prediction can be immediately confirmed or aborted, and more input has to be read to reject the wrong prediction. However, the prediction made during the open $p\overline{q}$ segment has to agree with the prediction at the adjacent $p\overline{q}$ singular point (case 3-(c) of Lemma 3.7), so if the tester was in location s_1 predicting u, transition $s_1 \rightarrow s_0$ is taken and if the tester was in location s_3 predicting $\overline{u}$, transition $s_3 \rightarrow s_2$ is taken. The only input signals which lead to two infinite runs are those that end with an infinite $p\overline{q}$ segment and they violate $p\mathcal{U}q$. To reject the wrong run which predicted u all along the segment, we forbid the tester to remain forever in s_1 without taking any transition by setting all transitions and all locations except s_1 as accepting. It is not hard to see that the tester is non-blocking and that every run satisfies Lemma 3.5 and Lemma 3.7 and hence it realizes the semantics of *until*, which concludes the proof of Proposition 8.2. □

8.2 Temporal Testers for $\diamondminus_{(0,a)}\, p$ and $\Diamond_{(0,a)}\, p$

Proposition 8.3. *One can construct a temporal tester that realizes* $\chi^{\diamondminus_{(0,a)}\, p}$.

Intuitively, the temporal tester for $\diamondminus_{(0,a)}\, p$ should monitor the truth value of p and memorize, using clocks, the times that this value has changed. As we shall see, a single clock is sufficient for this tester. Let $u = \chi^{\diamondminus_{(0,a)}\, p}(w)$. When p holds in w for some interval I with endpoints t_i and t_j, then $u = 1$ in the *open* interval $I \oplus (0,a) = (t_i, t_j + a)$

regardless of whether I is of the form (t_i, t_j), $(t_i, t_j]$, $[t_i, t_j)$ or $[t_i, t_j]$ (see Table 3.1). In other words, the value of u at t_i and $t_j + a$ does not depend on the value of p at t_i and t_j, respectively.

Throughout open segments where p holds, u is true (for every t in the segment, there is $t' < t$ in the same segment where p is true). Suppose now that p becomes false at t_1 until t_2. There are three possibilities depending on the duration $t_2 - t_1$ of the $\overline{p}$ segment:

1. $t_2 - t_1 < a$ (this includes, of course, the case where $t_1 = t_2$ is a singular point). In that case, for any t between t_1 and t_2, there is $t' \in t \ominus (0, a)$ which is smaller or equal to t_1 and where p holds, hence the value of u remains true throughout the $\overline{p}$ segment. This case is illustrated in Figure 8.3-(a).
2. $t_2 - t_1 = a$ and hence $t - a < t_1$ for any $t \in (t_1, t_2)$, and there is $t' \in (t - a, t_1]$ where p holds and the property is satisfied for all such t. At time t_2, the "previous" time where p was true is at $t_2 - a = t_1$ (or its left neighborhood), while the operator requires such existence within $(t_2 - a, t_2)$, hence the property is violated at t_2. Consequently, u is true in (t_1, t_2) and false at t_2. This case is illustrated in Figure 8.3-(b).
3. $t_2 - t_1 > a$. Then u is true throughout $(t_1, t_1 + a)$ (see the previous observation) and false in $[t_1 + a, t_2]$ because for any $t \in [t_1 + a, t_2]$, $t \ominus (0, a)$ is within (t_1, t_2) and p is false throughout that interval (see Figure 8.3-(c)).

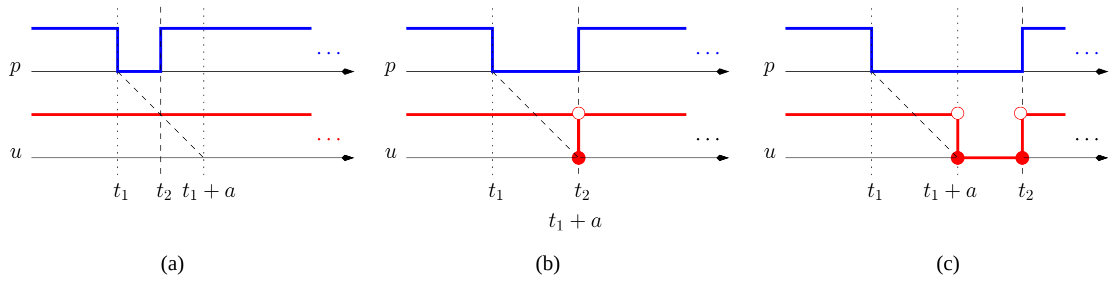


Fig. 8.3. Signal where p does not hold between t_1 and t_2 (a) $t_2 - t_1 < a$, (b) $t_2 - t_1 = a$ and (c) $t_2 - t_1 > a$

The temporal tester depicted in Figure 8.4 observes the behavior w and moves through locations $\{s_0, s_1, s_2\}$ generating the output (see Figure 8.5 for an illustration of a run). At time 0, the output is trivially $\overline{u}$. In location s_0, the tester reads a p-segment and outputs u. Singular occurrences of $\overline{p}$ (transition $s_0 \longrightarrow s_0$) are ignored (the output at these singular points remains u). When the input behavior w switches to $\overline{p}$ the tester moves from s_0 to s_1 and the clock x is reset (as we have seen, the tester does not distinguish whether the input was still p or already $\overline{p}$ at the moment of the transition, hence $s_0 \longrightarrow s_1$ is labeled by any letter). The clock x measures the distance from the latest occurrence of p and as long as its value is smaller than a, the output remains u. From location s_1, there are three possible continuations, that correspond exactly to the 3 possible relations between the duration of $\overline{p}$ and a:

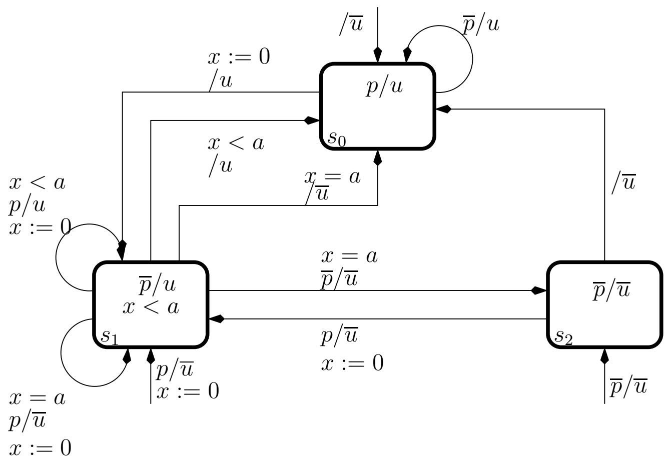


Fig. 8.4. The temporal tester for $\diamondminus_{(0,a)}\, p$

1. p occurs before x reaches a meaning that the segment of $\overline{p}$ had the duration strictly smaller than a (transitions $s_1 \longrightarrow s_1$ and $s_1 \longrightarrow s_0$ with guard $x < a$). Such "short" periods of $\overline{p}$ are ignored by the tester, and the output remains continuously u.
2. p occurs when x reaches a. This situation is realized by transitions $s_1 \longrightarrow s_0$ and $s_1 \longrightarrow s_1$ with guard $x = a$. The output at the transition is $\overline{u}$.
3. x reaches a while p continues to be false, the tester moves to s_2. The output label of the transition is $\overline{u}$. In location s_2 the tester outputs $\overline{u}$ since the value of x is strictly greater than a, meaning that the previous occurrence of p happened more than a time ago. If the tester observes p either as a singular point or as a segment, transition $s_2 \longrightarrow s_1$ or $s_2 \longrightarrow s_1$ are taken, respectively. The output label of the transition is $\overline{u}$.

Note that independently of the input values, the output signal $u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdots$ is of the form where all $\dot{u}_i = 0$, that is all the intervals where u is false are closed and intervals where u is true are open (following the observation that $I \oplus (0,a)$ is an open interval regardless of the form of I), and all the positive open segments $(u_i)_{r_i} = 1$ have the minimum duration $r_i \geq a$ (following Lemma 3.2).

Proposition 8.4. *One can construct a temporal tester that realizes* $\chi^{\diamond_{(0,a)}\, p}$.

The temporal tester for $\diamond_{(0,a)}\, p$ is similar to the one of $\diamondminus_{(0,a)}\, p$ and is shown in Figure 8.7. Unlike its past counterpart, it generates the output signal non-deterministically and checks whether the actual input confirms such predictions, aborting the wrong ones. Similarly to the past case, whenever p is true during some interval I, which can be of any of the type (t_i, t_j), $(t_i, t_j]$, $[t_i, t_j)$ or $[t_i, t_j]$, u is true throughout the *open* interval $I \ominus (0,a) = (t_i - a, t_j)$ (see Table 3.1). When p becomes false, there are three different cases concerning the duration of the $\overline{p}$ interval I with endpoints t_1 and t_2. To avoid repetition, we just illustrate these cases in Figure 8.6.

When an open segment labeled by p is observed, the tester is in location s_0 and the output is u throughout that segment. This prediction is immediately confirmed because for any t in the open p-segment there is $t' > t$ in the same segment such that p holds at t'.

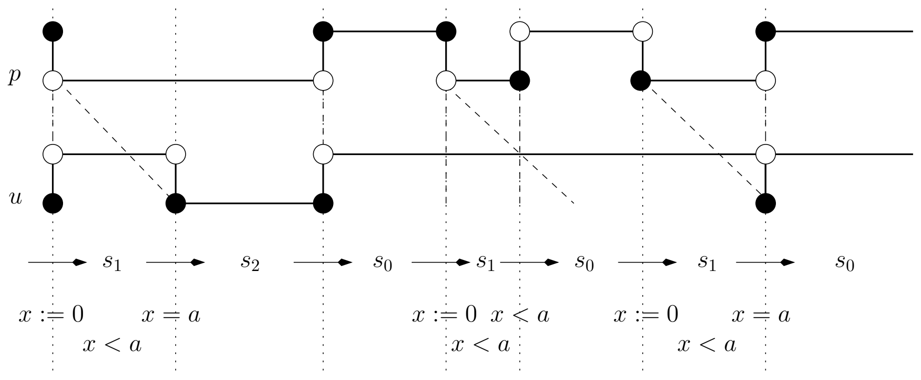


Fig. 8.5. A behavior of the temporal tester for $\diamondminus_{(0,a)} p$ on a given input signal.

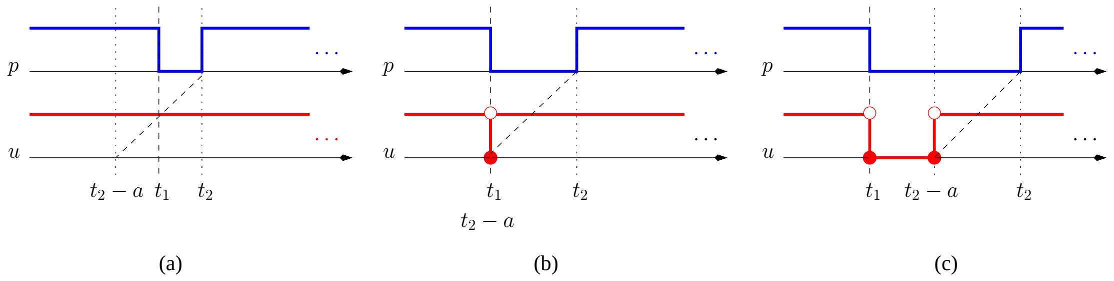


Fig. 8.6. Signal p that does not hold between t_1 and t_2; (a) $t_2 - t_1 < a$ (b) $t_2 - t_1 = a$ and (c) $t_2 - t_1 > a$

Singular occurrences of $\overline{p}$ are ignored ($s_0 \longrightarrow s_0$). When the input behavior w becomes false at t_1 until t_2, the tester can make one of the three different predictions, which correspond to the 3 possible relations between $t_2 - t_1$ and a:

1. Predict that the duration $t_2 - t_1$ of the $\overline{p}$ segment will be smaller than a. This situation is realized by location s_2, which is entered after the last occurrence of p. At the incoming transitions, a clock x which measures the distance between two consecutive occurrences of p is reset. The tester has to observe p before x reaches a (transitions $q_2 \longrightarrow q_2$, $q_2 \longrightarrow q_1$, $q_2 \longrightarrow q_3$ or $q_2 \longrightarrow q_0$). The run is aborted if x reaches a before p is observed. The output at location s_2 remains continuously u.
2. Predict the duration of the $\overline{p}$ segment to be exactly a. The tester moves to location s_1 which outputs u and the output at the transition is $\overline{u}$. The clock x is reset upon entering s_1 and the prediction is confirmed only if the next occurrence of p arrives when $x = a$ (transitions $q_1 \longrightarrow q_1$, $q_1 \longrightarrow q_2$, $q_1 \longrightarrow q_3$ and $q_1 \longrightarrow q_0$), otherwise the run is aborted.
3. Finally, the tester can predict that the duration of the $\overline{p}$ segment will be greater than a. The tester moves to s_3 which outputs $\overline{u}$ and the incoming transition is labeled by $\overline{u}$. It has to guess non-deterministically the time instant t such that $t + a$ is the last $\overline{p}$ time instant. At that time the tester moves to s_1 where, after a time the prediction is confirmed or the run is aborted.

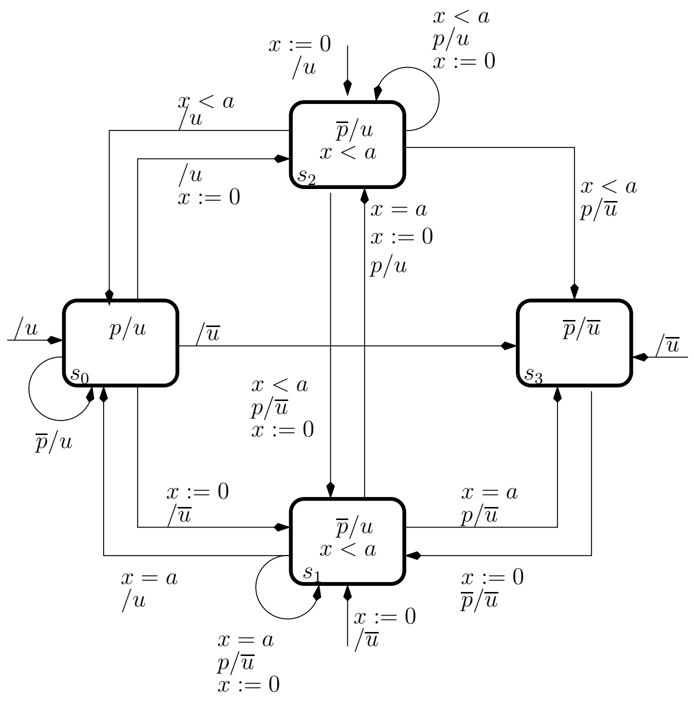


Fig. 8.7. The temporal tester for $\diamondsuit_{(0,a)}\, p$

Figure 8.8 illustrates some of the runs of the tester on an input signal of the form $w = w' \cdot p^{r_0} \cdot \dot{\overline{p}} \cdot \overline{p}^{r_1} \cdot \dot{\overline{p}} \cdot p^{r_2} \cdot w''$. The signal is false between t and t' with $t' - t > a$. At time t, the tester enters s_1, s_2 or s_3 depending whether it predicts that $t' - t = a, t' - t < a$ or $t' - t > a$, respectively. Runs that lead to s_1 and s_2 at t are aborted at most at $t + a$, because at that time the tester still observes a $\overline{p}$-segment, contrary to the prediction. The only correct prediction at t is to move to s_3. From s_3, the tester has to "guess" the time $t' - a$ to move to s_1. Predicting this transition at some other time leads to an eventual abortion of the run.

8.3 Temporal Testers for $\ominus_a p$ and $\diamondsuit_a p$

The operators $\ominus_a p$ and $\diamondsuit_a p$ are shift operators and, in general, may need infinitely many states and clocks. Nevertheless, when their input is restricted to signals with bounded variability, they can be realized by timed automata. Making use of Proposition 3.3 we apply this operation to signals w such that $w = \chi^{\ominus_I p}(w')$ or $w = \chi^{\diamondsuit_I p}(w')$ with I of the form $(0, d), (0, d], [0, d)$ or $[0, d]$. Such signals, according to Lemma 3.2, have the property that for any decomposition of w, for every segment $(w_i)_{r_i}$ such that $w_i = 1$, $r_i \geq d$, hence the number of changes that they may exhibit in an interval of duration a is bounded.

Proposition 8.5. *One can construct a temporal tester that realizes $\chi^{\ominus_a p}$ relative to input signals that satisfy the bounded variability assumption of Lemma 3.2.*

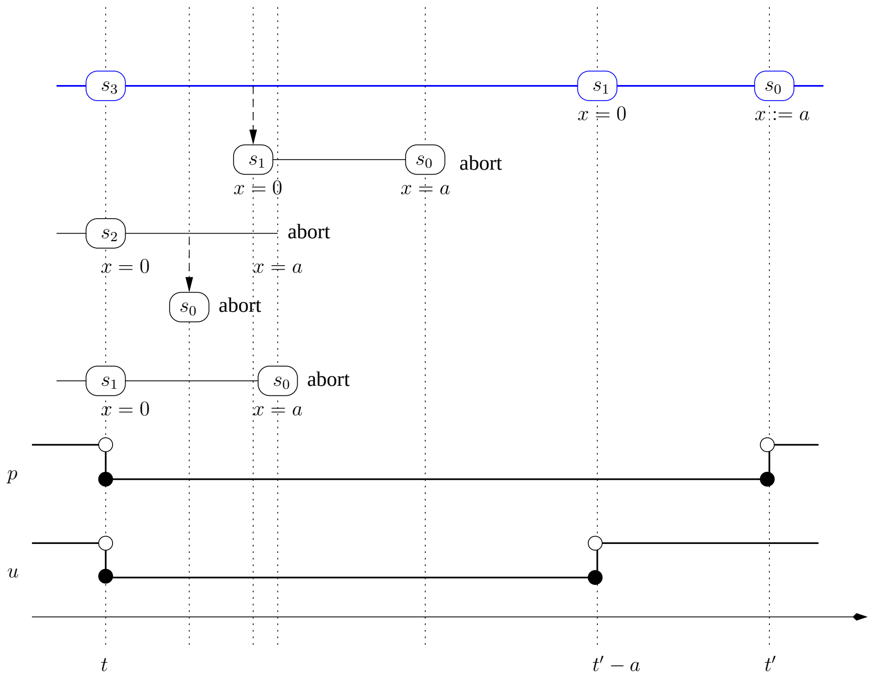


Fig. 8.8. Some of the behaviors of the temporal tester for $\Diamond_{(0,a)}\, p$ on an input signal. Only the upper run is not aborted.

We decompose the tester into two components, the *input observer* and the *output generator*, illustrated in Figure 8.9-(a) and 8.9-(b), respectively. The observer realizes a kind of continuous shift register which memorizes the value of the input signal in a past temporal window of length a. Signals satisfying the bounded variability property (Lemma 3.2) will have at most $2n$ changes in any such temporal window, with $n = \lceil \frac{a}{d} \rceil$. Hence these changes can be memorized with $2n$ clocks $\{x_1, y_1, \ldots, x_n, y_n\}$ that measure the time elapsed since subsequent changes in the signal values, and $2n$ Boolean variables $\{p_1^x, p_1^y \ldots, p_n^x, p_n^y\}$ that specify the values of the signal at the singular points.

The input observer reads the bounded variability input signal w and memorizes the relevant changes in the signal. Initially all the clocks are set to be inactive. Clocks x_i and y_i measure the time from the beginning of the i^{th} segment labeled by $\overline{p}$ and p, respectively, within the temporal window of length a, and p_i^x, p_i^y memorize the value of p at the singular end points of the corresponding segment. The input observer consists of $2n$ locations that we encode using two states, s_0 and s_1, and a counter i of bounded size n. We use (s, i) to denote these locations. Initially, all clocks are set to be inactive and the tester moves to $(s_0, 1)$ if the first open segment in the input signal is $\overline{p}$, or to $(s_1, 1)$ if the input signal starts with an open p-segment. From location (s_0, i), the observer moves to (s_1, i) when p becomes true, resets the clock y_i and assigns to p_i^y the value of p at the moment of transition . Note that we cannot have a decomposition $w = w' \cdot 0 \cdot \dot{1} \cdot 0 \cdot w''$ because of the bounded variability assumption. When in (s_1, i), two situations may occur, either p becomes false for some positive period in time, hence the tester moves to s_0 incrementing the counter $i := i + 1$, and setting the clock x_i and variable p_i^x, or the tester observes $\overline{p}$ at a singular point, followed by another p segment (there is no bounded variability assumption on the duration of $\overline{p}$-segments). In that case, the observer increments the counter, sets p_i^x and resets *both* clocks x_i and y_i.

Whenever the clock y_1 reaches a, it is guaranteed that the changes memorized by x_1 and y_1 have been taken into account by the output generator described below, and the two clocks do not influence the future values of u, hence they can be discarded. To keep the number of clocks bounded, we recycle clocks by applying the operation sh which consists in shifting the values of clock and Boolean variables $x_i := x_{i+1}$, $y_i := y_{i+1}$, $p_i^x := p_{i+1}^x$ and $p_i^y := p_{i+1}^y$ for all i and decrementing the counter $i := i - 1$. This operation guarantees that the counter i always remains bounded. Moreover, with this operation the clocks x_1 and y_1 represent at any time $t + a$ the time elapsed since the oldest "active" change in p within the interval $[t, t + a]$.

The output generator uses clocks x_1, y_1 and variables p_1^x, p_1^y to produce the value of u. Initially, at location s_{in} the generator trivially outputs $\overline{u}$ during the interval $[0, a)$. When x_1 reaches a and $y_1 < a$, this is the beginning of an a-shifted past $\overline{p}$-segment and the generator moves to $s_{\overline{u}}$ and outputs $\overline{u}$ as long as $y_1 < a$. If both x_1 and y_1 reach a simultaneously, this means that the $\overline{p}$-segment was, at most, punctual, and the automaton moves to s_u and outputs u. From both states $s_{\overline{u}}$ and s_u, the condition $y_1 = a$ ($x_1 = a$, respectively) indicates the end of the current segments and triggers a transition. The values of the output at singular points are based on values memorized in p_1^y and p_1^x. As we can see, the generator outputs the signal u that corresponds exactly to the input signal w shifted by a, that is $w[t] = u[t + a]$. □

Proposition 8.6. *One can construct a temporal tester that realizes $\chi^{\Diamond_a p}$ relative to input signals w that satisfy the bounded variability assumption of Lemma 3.2.*

The temporal tester for $\Diamond_a p$ is very similar to the past operator. It is decomposed as well into an input observer and an output generator but due to the acausality of the operator, the operation mode is slightly different. To avoid repetition, we explain the main differences with respect to the tester for the past operator and depict the generator and the observer in Figures 8.10-(a) and 8.10-(b), respectively.

The output generator has to produce at time t values that can be confirmed only at $t+a$. Hence, all the responsibility on maintaining the variability of the output bounded is delegated to the output generator[1] (condition $y_i \geq d$ on all the transitions outgoing from s_u). More importantly, the memorization should now apply to the output: clocks x_i, y_i and variables u_i^x, u_i^y are reset by the generator[2] as it changes its output in order to represent the predicted signal. It is also responsible for taking the sh transition when $y_1 = a$ and the oldest event in the predicted output has already been confirmed or contradicted.

The role of the observer is now to compare the input signal with those predictions. It may move between states s_0 and s_1 according to the values of the input and the values

[1] For predictions of the form $u = \dot{1} \cdot 1^r \cdot u'$, the duration of the first u-segment can be arbitrarily small. Hence, we have an additional location s_a that generates the first u-segment without the condition $y_i \geq d$ on the outgoing transitions.

[2] We use the assignment $u_i^y := \{0, 1\}$ as syntactic sugar to collapse two transitions into one and represent the fact that the predicted output can be either $\overline{u}$ or u at the singular point, and the predicted value is memorized.

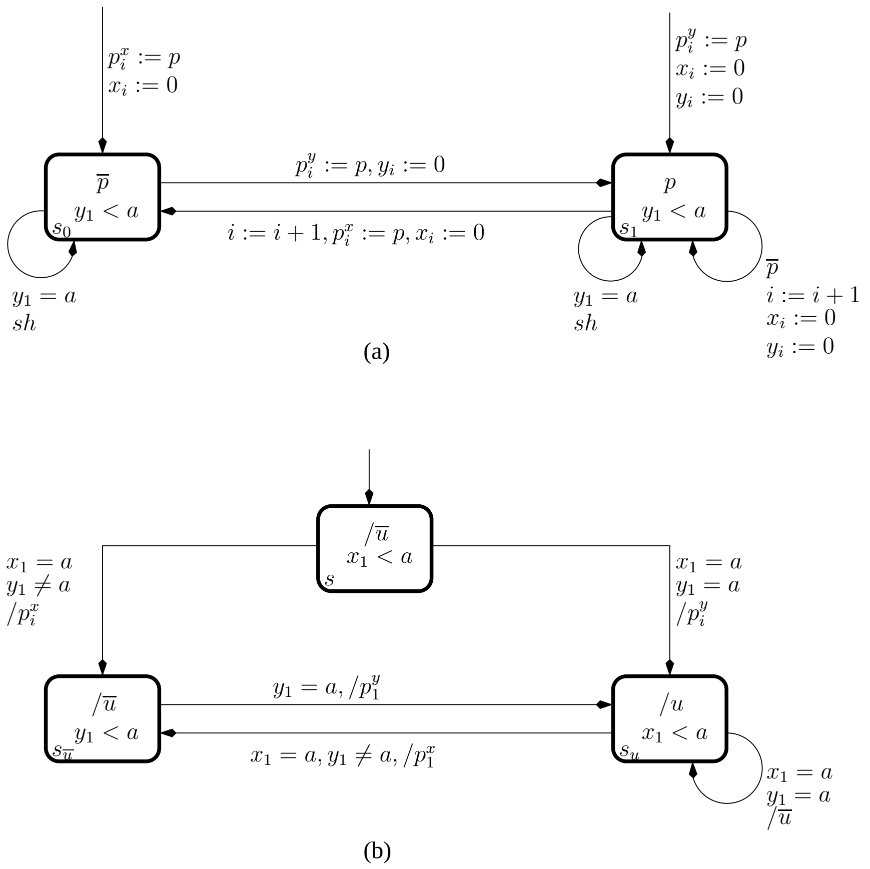


Fig. 8.9. The temporal tester for $\diamondminus_a p$ - (a) input observer (b) output generator

of the clocks that correspond to the form of the predicted output. When those disagree the run is aborted. It is not hard to see that the generator produces valid outputs whose features are memorized and that the observer checks the conformity of these predictions with the input and as result we have $u[t] = w[t+d]$. $\square$

To complete the construction for MITL we just need to compose the testers for the propositional, untimed and timed operators according to the structure of the formula. The parallel composition of transducers is fairly standard and we give only the definition of an input/output composition of signal transducers $\mathcal{A}^1 \triangleright \mathcal{A}^2$ where the output of $\mathcal{A}^1$ is the input of $\mathcal{A}^2$. Note that the generalized Büchi condition comes from such a composition of testers for unbounded operators as we need to identify accepting runs of $\mathcal{A}^2$ triggered by outputs of accepting runs of $\mathcal{A}^1$.

Definition 8.7 (I/O Composition). *Let $\mathcal{A}^1 = (\Sigma^1, \Gamma^1, Q^1, \mathcal{C}^1, I^1, \Delta^1, \lambda^1, \gamma^1, q_{in}^1, \mathcal{F}^1)$ and $\mathcal{A}^2 = (\Sigma^2, \Gamma^2, Q^2, \mathcal{C}^2, I^2, \Delta^2, \lambda^2, \gamma^2, q_{in}^2, \mathcal{F}^2)$ be timed signal transducers such that $\Gamma^1 = \Sigma^2$. Their I/O composition is the transducer*

$$\mathcal{A} = \mathcal{A}^1 \triangleright \mathcal{A}^2 = (\Sigma^1, \Gamma^2, Q, \mathcal{C}, I, \Delta, \lambda, \gamma, q_{in}, \mathcal{F})$$

where

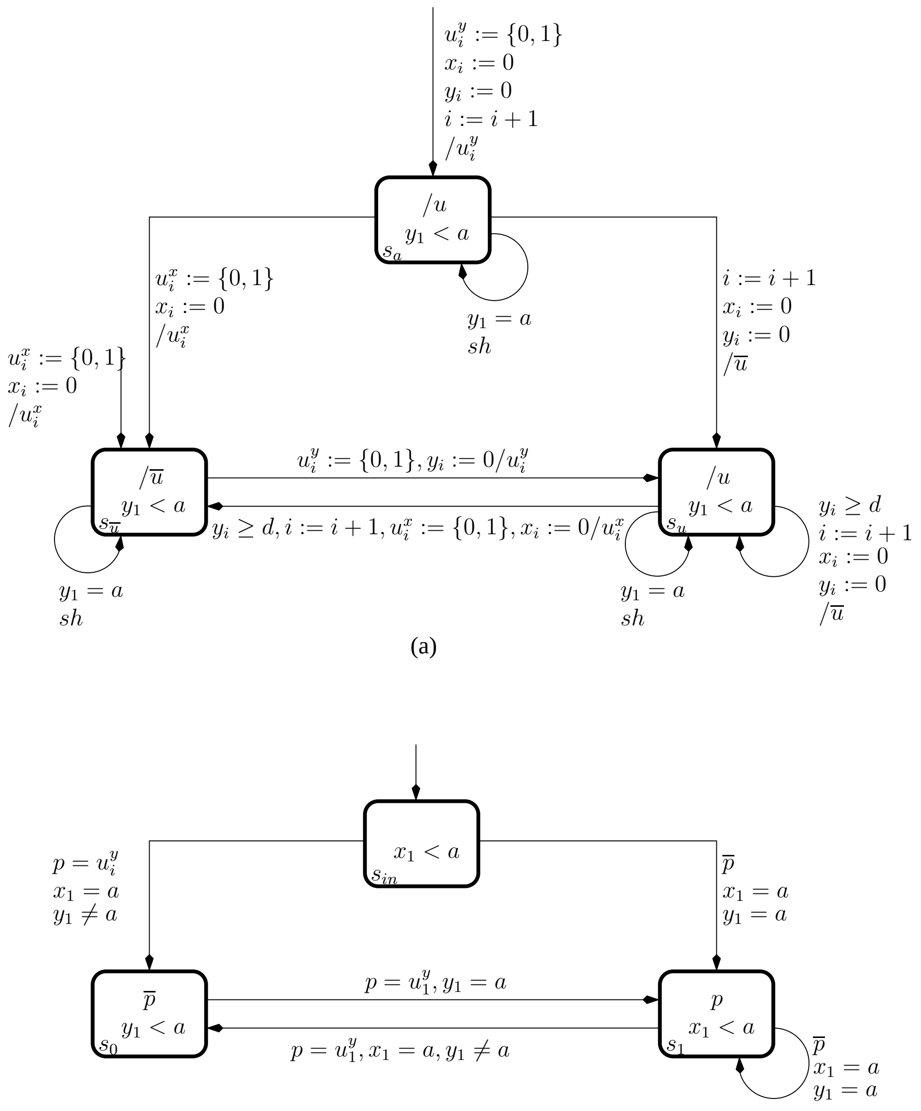


Fig. 8.10. The temporal tester for $\Diamond_a p$ - (a) output generator (b) input observer

$$Q = \{(q^1, q^2) \in Q^1 \times Q^2 \text{ s.t. } \gamma^1(q^1) = \lambda^2(q^2)\},$$

$C = C^1 \cup C^2, \lambda(q^1, q^2) = \lambda^1(q^1), \gamma(q^1, q^2) = \gamma^2(q^2)$ and $I_{(q^1,q^2)} = I^1_{q^1} \cap I^2_{q^2}$. The transition relation Δ is the restriction to Q of the set of all transitions of either of the following forms[3]

$$\begin{array}{ll}
\delta_{12} : ((q^1, q^2), g^1 \cap g^2, \rho^1 || \rho^2, (q'^1, q'^2)) & \textit{and } \lambda(\delta_{12}) = \lambda^1(\delta^1), \gamma(\delta_{12}) = \gamma^2(\delta^2) \\
\delta_1 : \ ((q^1, q^2), g^1 \cap I_{q^2}, \rho^1, (q'^1, q^2)) & \textit{and } \lambda(\delta_1) = \lambda^1(\delta^1), \gamma(\delta_1) = \gamma^2(q^2) \\
\delta_2 : \ ((q^1, q^2), g^2 \cap I_{q^1}, \rho^2, (q^1, q'^2)) & \textit{and } \lambda(\delta_2) = \lambda^1(q^1), \gamma(\delta_2) = \gamma^2(\delta^2)
\end{array}$$

[3] When in initial state (q^1_{in}, q^2_{in}), the two transducers need to take the joint transition

such that $\delta^1 = (q^1, g^1, \rho^1, q'^1) \in \Delta^1$ and $\delta^2 = (q^2, g^2, \rho^2, q'^2) \in \Delta^2$.
The accepting sets are defined as follows. Let $\mathcal{F}^1 = \{F_1^1 \dots F_n^1\}$ and $\mathcal{F}^2 = \{F_1^2 \dots F_m^2\}$. Then $\mathcal{F} = \{F_1^{1'} \dots F_n^{1'}, F_1^{2'}, \dots F_m^{2'}\}$ where each $F_i^{1'} \in \mathcal{F}$ consists of locations (q_1, q_2) such that $q_1 \in F_i^1$, transitions of the form δ_{12}, δ_1 such that $\delta^1 \in F_i^1$ and transitions of the form δ_2 such that $q^1 \in F_i^1$. Similar rules apply to locations and transitions that are in sets $F_i^{2'} \in \mathcal{F}$.

It is not hard to see that $\mathcal{A}^1 \triangleright \mathcal{A}^2$ realizes the sequential function obtained by composing the sequential functions realized by $\mathcal{A}^1$ and $\mathcal{A}^2$.

Corollary 8.8 (Main Result). MITL *formulae can be transformed into timed automata using a modular procedure.*

8.4 Discussion

In this section, we discuss some work related to our translation of MITL formulae to timed automata. The decidability of MITL was established in [AFH96], which gives a tableau-like procedure for translating MITL formulae to timed automata. This version of MITL contained only future temporal operators.

An investigation of past and future versions of MITL was carried out in [AH92b] using two-way timed automata, that is, automata having the ability to change the direction of reading. The authors describe a strict hierarchy of timed languages based on the number of direction reversals needed to recognize them (which roughly corresponds to the nesting depth of past and future operators).

Event-recording automata, where only the time of the *last* occurrence of every input letter can be remembered by a clock, have been shown to be determinizable in [AFH99]. Event-clock automata, introduced in the same paper, constitute a generalization of the latter which allow also “event-predicting” clocks, to express the acausality of future temporal operators. In [HRS98, RS97], the authors introduce *event-clock temporal logic* ECL and show that it is expressively equivalent to MITL with future and past. The results of [HRS98] provide an alternative indirect route to translate MITL formulae with future and past to timed automata. First the MITL formula is transformed into an ECL formula, which can be translated to an equivalent event-clock automaton, from which one can obtain the corresponding timed automaton.

Finally, we also mention our previous translation of past [MNP05] and future [MNP06] MITL formulae to timed automata using temporal testers. In [MNP05, MNP06], the definitions of the logic and signals differ from [AFH96] and this thesis in the following respects:

1. We disallow signals that admit punctuality and restrict ourselves to right-continuous signals, namely those that can be decomposed into a sequence of left-closed right-open segments;
2. We restrict the temporal logic to closed intervals;
3. We modify the semantics of $p\mathcal{U}q$ to require a moment where both p and q hold.

The restriction to non-punctual signals seems reasonable from a semantic point of view, an the two other modifications are consequences of this choice as we want the output of the testers to be valid signals as well. The restriction to right-continuous signals simplifies significantly the construction of testers as no special treatment is required for the input/output symbols on transitions. This simplicity is expressed in the construction of the testers for $\diamondminus_a$ and $\diamond_a$. Memorizing the form of a right-continuous signal with n segments requires $O(n)$ locations. In our construction for the general case we need $O(2^n)$ states for all possible values at singular points (variables p_i^x, p_i^y). The main limitation of the restricted logic is the inability to specify events (such as the rising and falling of a signal) which prevents, for example, expressing properties such as bounded variability. The construction presented in this thesis completes our previous results by considering MITL formulae and signals in their most general form, and providing a unified translation of MITL formulae with past, future and events to timed automata.

9

Conclusions

This thesis was motivated by a very practical concern: improving the design process for analog and mixed-signal circuits by introducing property-based monitoring of analog signals based on temporal logic. Although practically motivated and geared toward industrial standards, tools and case-studies, this work did not neglect the underlying theoretical foundations. On the contrary, this thesis shows that starting from rigorous studies of the semantics of timed systems, one can build (prototypes of) industrial-strength tools.

Below we summarize what we consider to be the major achievements of this thesis on the theoretical and practical sides.

Theory: the study of the marking procedure for monitoring has led to a point of view on satisfaction of sub-formulae which finally converged with the powerful idea of timed testers. We strongly believe that the tester-based translation from MITL to automata described in Chapter 8 is the clearest explanation to date concerning the relation between the two formalisms, the roles of future and past operators, the influence of bounded variability and the origins of non-determinism in timed automata. Our definitions of timed transducers and their runs over signals in a segment-point decomposition, allow us to realize input-output operators over such signals in a neat way. Finally, we mention the idea of transforming a bounded future MITL formula into a past (and hence causal) formula for the purpose of controller synthesis.

Practice: this thesis provided a pioneering contribution to the verification of analog circuits. We suggested a specification formalism, monitoring algorithms and a comprehensive prototype tool for performing this task. The feedback of those in the semiconductor and EDA industries who came to know the methodology and the tool was extremely positive, which may give hope for an eventual industrial transfer of these results. A large part of this success is due to the demonstration of the applicability of this approach via real-life case studies.

Some of the future work directions inspired by this thesis are described below:

1. Extending the scope of the AMT tool by providing a richer language and additional types of queries. Among the extension we mention: the expression of non-temporal properties (frequency domain, for example), interactions which are more complex than pointwise Booleanization between real-valued and Boolean signals, extraction

of parameters and other quantitative measures (compared to the purely yes/no nature of the current answers).

2. Tighter integration between the AMT tool and existing simulators which will facilitate efficient event detection, combination with test generation methods and utilization of building blocks that already exist in the simulators.
3. Extend the construction of temporal testers to cover some subset of the timed regular expressions of [ACM02]. This task is particularly important because regular expressions have a special importance in SVA [Acc04].
4. Gain a better understanding of the origins of non-determinism in timed automata and the situations where one can get rid of it without auxiliary assumptions such as bounded variability.
5. Complete and optimize the implementation of the translation from MITL to timed automata and use it for model checking of timed systems within the IF toolset.

A

On Synthesizing Controllers from Bounded-Response Properties

A.1 Introduction

The problem of synthesizing controllers automatically from high-level specifications has been posed by Church [Chu63] and solved theoretically by Büchi and Landweber [BL69, TB73]. Although the topic has been subject to further, more modern, investigations, synthesis has not enjoyed the passage from theory to practice as did the similar and simpler problem of verification, mostly due to the practical complexity of the proposed algorithms. Recently some improvements have been made for untimed [PPS06, PP06] and timed [CDF$^+$05] systems, that led to the synthesis of some non trivial controllers. This work is a further step in this direction which attempts to give a general feasible solution for the following problem:

Given a bounded-response temporal property φ defined over two distinct action alphabets A and B (encoded using mutually-disjoint sets of propositional variables), build a finite-state transducer (controller) from A^ω to B^ω such that all of its behaviors satisfy φ at all positions.

The controller in question is realized by an automaton that observes what the environment does (some $a \in A$), changes its state accordingly and outputs some $b \in B$. The whole situation can be viewed as a two-player zero-sum game between the controller and its environment where one seeks a winning strategy for the controller (see [Mal07] for a unified game-theoretic model). Unlike other approaches, for example those used in the control of discrete event systems [RW89] or our previous work [MPS95, AMP95], we do not start with a given “plant” or “arena” in a form of a transition system and an acceptance/winning condition expressed in terms of its states. Our starting point, like in [PR89], is a temporal logic formula which specifies constraints on the behaviors of the players as well as desired properties of their interaction. Hence the first step in the synthesis procedure is to derive the automaton *from the formula* and then apply synthesis algorithms to this automaton.

A major difficulty in such a procedure stems for the fact that synthesis algorithms are more naturally defined over *input-deterministic* automata, or, to be more precise, over automata where each non-deterministic choice can be *unambiguously* attributed to one of the two players. In such automata each joint choice of the two players induces only one transition from every state.[1] In contrast, the commonly-used procedures for translating temporal logic formulae go through non-deterministic automata whose determinization leads to automata of prohibitively-large size. Another obstacle toward the efficient realization of synthesis algorithms is the fact that the acceptance conditions in the generated automata require a complicated fixed-point computation in order to find the winning states and strategies.

In this work we avoid some of these problems by restricting our attention to *bounded-response* properties which are known to be equivalent to safety properties. These properties represent a large part of what users are interested in (especially in hard real-time systems) and lead to automata with simpler acceptance conditions (just avoid bad states) and hence to a simpler synthesis procedure. Concerning the limited scope of bounded-response properties compared to more general *liveness* properties, we can make the following comments. Liveness properties typically specify something that should "eventually" happen without specifying an upper bound on the time to elapse between now and that eventuality. Obviously, liveness properties can be viewed as an abstraction of the real specification which requires not only that some response is eventually forthcoming (which is often useless by itself), but also provides an *upper bound* on the maximal delay on the arrival of the response. In some cases, the use of such abstractions may be justified on various grounds. However, we hope to convince the reader that, in many other cases, the synthesis from bounded-response properties is very relevant and preferable and can be carried out efficiently for non-trivial problems. For such cases, why settle for an abstraction when you can work directly with the precise specification?

The main contribution of this paper is an efficient machinery that allows one to synthesize controllers automatically from specifications expressed using the real-time temporal logic MTL [Koy90], often interpreted of the time domain $\mathbb{R}_+$. Our first contribution is a transformation of such formulae, under *bounded variability assumptions* to *deterministic* timed automata. This determinization is of particular interest as it is based on transforming the formula into a *past* formula and then applying the ideas presented in [MNP05]. The obtained automaton is then interpreted as a timed game automaton [MPS95, AMP95] to which we apply a synthesis algorithm to derive the controller.

The rest of the paper is organized as follows: Section A.2 presents the syntax and semantics of the bounded-response fragment of MTL. Section A.3 shows how to translate future bounded MTL formulae into past formulae and deterministic timed automata. Section A.4 reports some preliminary experiments in synthesizing an arbiter from its specifications, while Section A.5 mentions ongoing and future efforts to improve the performance.

A.2 Signals and their Bounded Temporal Logic

Timed behaviors can be described using either *time-event sequences* consisting of instantaneous events separated by time durations or discrete-valued *signals* which are

[1] A notable exception is the case where the controller has limited observability and thus, after observing a sequence of adversary actions it may find itself in one of several states and its chosen action should be good with respect to *all* these states. In this case, the nondeterminism plays in favor of the adversary.

functions from time to some discrete domain. In this work we use Boolean signals as the semantic domain, but the extension of the results to time-event sequences (which are equivalent to the timed traces of [AD94]) need not be a difficult exercise.

Let the time domain $\mathbb{T}$ be the set $\mathbb{R}_{\geq 0}$ of non-negative real numbers and let $\mathbb{B} = \{0, 1\}$. An n-dimensional Boolean signal ξ is a partial function $\xi : \mathbb{T} \to \mathbb{B}^n$ whose domain of definition is an interval $I = [0, r)$, $r \in \mathbb{N} \cup \{\infty\}$. We say that the length of the signal is r and denote this fact by $|\xi| = r$ and let $\xi[t]$ stand for the value of the signal at time t. We use $t \oplus [a, b]$ to denote $[t + a, t + b]$, that is, the Minkowski sum of $\{t\}$ and $[a, b]$, and $t \ominus [a, b] = [t - b, t - a] \cap \mathbb{T}$ for the inverse operation with saturation at zero. In the sequel we will restrict our attention to well-behaving signals whose variability is bounded.

Definition A.1 (Bounded Variability). *A signal ξ is of (Δ, k)-bounded variability if for every interval of the form $[t, t + \Delta]$ the number of changes in the value of ξ is at most k. A bounded-variability signal is a signal for which such $\Delta > 0$ and finite k exist.*

Proposition A.2 (Preservation of Bounded Variability). *Let ξ_1 and ξ_2 be two infinite bounded variability signals characterized, respectively, by (Δ, k_1) and (Δ, k_2), and let $\xi = \xi_1 \; op \; \xi_2$ be a signal obtained by applying the Boolean operation op to ξ_1 and ξ_2. Then, ξ is of $(\Delta, k_1 + k_2)$-bounded variability.*

This fact, which follows from the observation that for ξ to switch at time t, at least one of ξ_1 and ξ_2 should switch, implies that if we assume bounded variability of the propositional signals, we will also have bounded variability for the signals that indicate the truth values of subformulae. Hence we can build the automaton corresponding to the formula in an inductive and compositional manner based on the temporal testers introduced in [KP05] for discrete time and extended in [MNP05, MNP06] for dense time. In this construction bounded variability will be guaranteed at all levels.

We define the logic MTL-B as a bounded-horizon variant of the real-time temporal logic MTL [Koy90], such that *all* future temporal modalities are restricted to intervals of the form $[a, b]$ with $0 \leq a \leq b$ and $a, b \in \mathbb{N}$, but allow the unbounded past operator $\mathcal{S}$ (*since*) which is not really unbounded. Note that unlike MITL [AFH96], we allow "punctual" modalities with $a = b$ and in this case we will use a as a shorthand for $[a, a]$. Another deviation from MTL is the introduction of an additional past operator *precedes* ($\mathcal{P}$) which is roughly the bounded *until* operator from the point of view of the *end* of the relevant segment of the signal. This operator is *not* proposed for user-friendliness purposes, but rather to facilitate the translation from future to past. The basic formulae of MTL-B are defined by the grammar

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 \mid \varphi_2 \mathcal{S}_{[a,b]} \varphi_1 \mid \varphi_2 \mathcal{S} \varphi_1 \mid \varphi_1 \mathcal{P}_{[a,b]} \varphi_2$$

where p belongs to a set $P = \{p_1, \ldots, p_n\}$ of propositions corresponding naturally to the coordinates of the n-dimensional Boolean signal considered. The *future fragment* of MTL-B uses only the $\mathcal{U}_{[a,b]}$ modality while the *past fragment* uses only the $\mathcal{S}_{[a,b]}$, $\mathcal{S}$ and $\mathcal{P}_{[a,b]}$ modalities. The satisfaction relation $(\xi, t) \models \varphi$, indicating that signal ξ satisfies φ

at position t, is defined inductively below. We use $p[t]$ to denote the projection of $\xi[t]$ on the dimension that corresponds to variable p.

$$\begin{array}{ll}
(\xi,t) \models p & \leftrightarrow p[t] = \mathrm{T} \\
(\xi,t) \models \neg\varphi & \leftrightarrow (\xi,t) \not\models \varphi \\
(\xi,t) \models \varphi_1 \vee \varphi_2 & \leftrightarrow (\xi,t) \models \varphi_1 \text{ or } (\xi,t) \models \varphi_2 \\
(\xi,t) \models \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 & \leftrightarrow \exists\, t' \in t \oplus [a,b]\ (\xi,t') \models \varphi_2 \text{ and} \\
 & \forall t'' \in [t,t'], (s,t'') \models \varphi_1 \\
(\xi,t) \models \varphi_2 \mathcal{S}_{[a,b]} \varphi_1 & \leftrightarrow \exists t' \in t \ominus [a,b]\ (\xi,t') \models \varphi_1 \text{ and} \\
 & \forall t'' \in [t',t], (\xi,t'') \models \varphi_1 \\
(\xi,t) \models \varphi_2 \mathcal{S} \varphi_1 & \leftrightarrow \exists t' \in [0,t]\ (\xi,t') \models \varphi_1 \text{ and} \\
 & \forall t'' \in (t',t], (\xi,t'') \models \varphi_1 \\
(\xi,t) \models \varphi_1 \mathcal{P}_{[a,b]} \varphi_2 & \leftrightarrow \exists t' \in t \ominus [0,b-a]\ (\xi,t') \models \varphi_2 \text{ and} \\
 & \forall t'' \in [t'-b,t']\ (\xi,t'') \models \varphi_1
\end{array}$$

It is important to note the difference between the future and the past operators (see Figure A.1): the *until* operator points from time t toward the future, while the *since* and *precedes* operators point from t backwards. On the other hand, the *until* and *precedes* operators differ from the *since* operators as they speak on the interval *before* the event that should be observed within a bounded time interval, while the latter refers to the interval immediately *after* its occurrence.

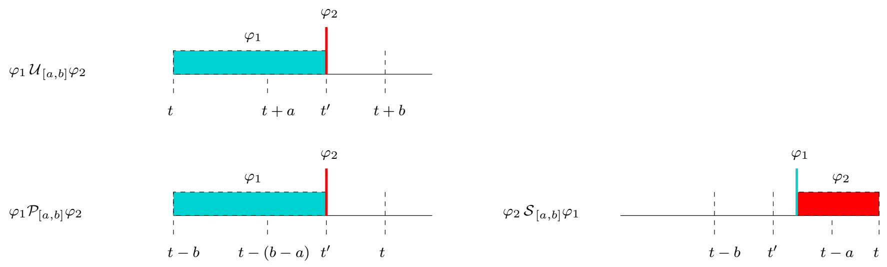

$\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$
φ_1
φ_2
t
$t+a$
t'
$t+b$
$\varphi_1 \mathcal{P}_{[a,b]} \varphi_2$
φ_1
φ_2
$t-b$
$t-(b-a)$
t'
t
$\varphi_2 \mathcal{S}_{[a,b]} \varphi_1$
φ_1
φ_2
$t-b$
t'
$t-a$
t

Fig. A.1. The semantic definitions of *until*, *precedes* and *since*.

From basic MTL-B operators one can derive other standard Boolean and temporal operators, in particular the time-constrained *sometime in the past*, *always in the past*, *eventually in the future* and *always in the future* operators whose semantics is defined as

$$\begin{array}{l}
(\xi,t) \models \diamondminus_{[a,b]} \varphi \leftrightarrow \exists t' \in t \ominus [a,b]\ (\xi,t') \models \varphi \\
(\xi,t) \models \boxminus_{[a,b]} \varphi \leftrightarrow \forall t' \in t \ominus [a,b]\ (\xi,t') \models \varphi \\
(\xi,t) \models \Diamond_{[a,b]} \varphi \leftrightarrow \exists t' \in t \oplus [a,b]\ (s,t') \models \varphi \\
(\xi,t) \models \Box_{[a,b]} \varphi \leftrightarrow \forall t' \in t \oplus [a,b]\ (\xi,t') \models \varphi
\end{array}$$

Note that our definition of the semantics of the timed *until* and *since* operators differs slightly from their conventional definition since it requires a time instant t' where *both* $(\xi,t') \models \varphi_2$ and $(\xi,t') \models \varphi_1$. For the untimed *since* operator we retain the standard semantics.

Each future MTL-B formula φ admits a number $D(\varphi)$ which indicates its *temporal depth*. Roughly speaking, to determine the satisfaction of φ by a signal ξ from any position t, it suffices to observe the value of ξ in the interval $[t, t+D(\varphi)]$. This property is evident from the semantics of the (bounded) temporal operators and admits the following recursive definition:

$$\begin{aligned}
D(p) &= 0\\
D(\neg\varphi) &= D(\varphi)\\
D(\varphi_1 \vee \varphi_2) &= \max\{D(\varphi_1), D(\varphi_2)\}\\
D(\varphi_1 \mathcal{U}_{[a,b]} \varphi_2) &= b + \max\{D(\varphi_1), D(\varphi_2)\}
\end{aligned}$$

Note that D is a syntax-dependent *upper bound* on the actual depth: the satisfiability of a formula φ may be determined according to segments of ξ shorter than $D(\varphi)$. For example, $D(\Box_{[a,b]}\,\mathrm{T}) = b$, but the formula requires no part of ξ for its satisfiability to be determined. At the end of the day we are interested in properties of the form $\Box\,\varphi$ where φ is any (future, past or mixed) MTL-B formula. These properties are interpreted over infinite-duration signals and require that all segments of ξ of length $D(\varphi)$ satisfy φ.

A.3 From MTL-B to Deterministic Timed Automata

In [MP04, MNP05] we have studied the relation between real-time temporal logics and deterministic timed automata. It turns out that the non-determinism associated with real-time logics has two rather *independent* sources described below.

- *Acausality*: the semantics of future temporal logics is acausal in the sense that the satisfiability of a formula at position t may depend on the value of the sequence or signal at time $t' > t$. If the automaton has to output this value at time t, it has no choice but to "guess" at time t and abort later at time t' the computations that correspond to wrong predictions (see more detailed explanation in [MNP06]). This bounded non determinism is harmless and in the untimed case, that is, for LTL, it can be determinized away. We conjecture that such a determinization procedure exists also for the timed case, but so far none has been reported. This problem does not exist for *past* temporal logic whose semantics is causal and hence it translates naturally into deterministic automata.
- *Unbounded variability*: when there is no bound on the variability of input signals, the automaton needs to remember the occurrence times of an unbounded number of events and use an unbounded number of clocks. All the pathological examples concerning non-determinizability and non-closure under complementation for timed automata [AD94] are based on this phenomenon.

In [MNP05] we have shown that the determinism of past MITL, compared to the non-determinism of future MITL, is a result of a syntactic accident which somehow imposes bounded variability (or indifference to small fluctuations) for the former but not the latter. The punctual version, past MTL, remains non deterministic (and of infinite memory) because the operator $\diamondminus_a$ realizes an ideal delay element which requires unbounded memory.

The approach taken in this work in order to get rid of both sources of non determinism is the following: we use full MTL, that is, allow punctual modalities, but assume that we are dealing with signals of (Δ, k)-bounded variability, hence we can dispense with the severe form of non determinism.[2] We then transform future MTL-B formulae to past MTL-B formula which, under the bounded variability assumption, can be translated to deterministic timed automata. This part of the result is an extension of what we have shown in [MNP05] for the (non-punctual) *since* operator.

The key idea of the transformation is to change the time direction from future to past and hence eliminate the "predictive" aspect of the semantics. We will present an operator Π which takes as an argument a future formula φ and a displacement d, and transforms it to an "equivalent" past formula ψ such that φ is satisfied by a signal from position t iff ψ is satisfied by the same signal from $t + d$.

Definition A.3 (Pastify Operator). *The operator Π on future* MTL-B *formulae φ and a displacement $d \geq D(\varphi)$ is defined recursively as:*

$$\begin{aligned}
&\Pi(p, d) = \diamondminus_d\, p\\
&\Pi(\neg\varphi, d) = \neg\Pi(\varphi, d)\\
&\Pi(\varphi_1 \vee \varphi_2, d) = \Pi(\varphi_1, d) \vee \Pi(\varphi_2, d)\\
&\Pi(\varphi_1 \mathcal{U}_{[a,b]} \varphi_2, d) = \Pi(\varphi_1, d-b) \mathcal{P}_{[a,b]} \Pi(\varphi_2, d-b)
\end{aligned}$$

Note that according the this definition $\Pi(\diamondsuit_{[a,b]}\, \varphi, d) = \diamondminus_{[0,b-a]}\, \Pi(\varphi, d-b)$.

Proposition A.4 (Relation between φ and $\Pi(\varphi, d)$). *Let φ be a bounded future formula and let $\psi = \Pi(\varphi, d)$ with $d \geq D(\varphi)$. Then for every ξ and $t \geq 0$ we have:*

$$(\xi, t) \models \varphi \textit{ iff } (\xi, t + d) \models \psi \tag{A.1}$$

Proof: We proceed by induction on the structure of the formula. The base case, the atomic propositions, satisfy (A.1) trivially. Proceeding to the inductive case, we show that if (A.1) holds for formulae with complexity (nesting of operators) m, it holds as well for formulae of complexity $m + 1$. For Boolean operators this is straightforward. Assume now that φ_1 and φ_2 satisfy (A.1) and we will show that so does $\varphi = \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$. Note that by definition, if $D(\varphi) = d$ then $D(\varphi_1) \leq d - b$ and $D(\varphi_2) \leq d - b$. Let $\psi_1 = \Pi(\varphi_1, d - b)$ and $\psi_2 = \Pi(\varphi_2, d - b)$. The fact the $(\xi, t) \models \varphi$ amounts to

$$\exists t' \in t \oplus [a, b]\ (\xi, t') \models \varphi_2 \wedge \forall t'' \in [0, t']\ (\xi, t'') \models \varphi_1.$$

According to the inductive hypothesis we have that for such t' and t''

$$(\xi, t' + d - b) \models \psi_2 \ \text{ and } \ (\xi, t'' + d - b) \models \psi_1.$$

By letting $r' = t' + d - b$ and $r'' = t'' + d - b$ and substituting the constraints on t' and t'' we obtain

[2] It is worth noting that the procedure of [Tri02] for subset construction on-the-fly, that is, determinization with respect to a *given* (and hence of bounded variability) input, works due to the same reasons.

$$\exists r' \in t + d \ominus [0, b - a]\ (\xi, r) \models \psi_2 \wedge \forall r'' \in [t + d - b, r]\ (\xi, r'') \models \psi_1,$$

which is exactly the definition of $(\xi, t + d) \models \psi_1 \mathcal{P}_{[a,b]} \psi_2$.

For the other direction assume $(\xi, t + d) \models \psi_1 \mathcal{P}_{[a,b]} \psi_2$ which means that

$$\exists r' \in t + d \ominus [0, (b - a)]\ (\xi, r') \models \psi_2 \wedge \forall r'' \in [t + d - b, r'](\xi, r'') \models \psi_1.$$

By the inductive hypothesis such r' and r'' satisfy

$$(\xi, r' - (d - b)) \models \varphi_1 \text{ and } (\xi, r'' - (d - b)) \models \varphi_1.$$

Letting $t' = r' - (d - b)$ and $t'' = r'' - (d - b)$ and substituting the constraints on r' and r'' we obtain

$$\exists t' \in t \oplus [a, b]\ (\xi, t') \models \varphi_2 \wedge \forall t'' \in [t, t']\ (\xi, t'') \models \varphi_1$$

which means that $(\xi, t) \models \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$. $\square$

Corollary A.5 (Equisatifaction of $\Box\, \varphi$ and $\Box\, \psi$). *An infinite signal ξ satisfies $\Box\, \varphi$ iff it satisfies $\Box\, \psi$ where $\psi = \Pi(\varphi, D(\varphi))$.*

We now proceed to the construction of a deterministic timed automaton accepting exactly signals satisfying a past MTL-B formula ψ under a bounded-variability assumption. The construction, inspired by [KP05], is compositional in the sense that it yields a network of deterministic signal transducers (testers), each corresponding to a subformula of ψ. The output of every tester for ψ' at time t equals to the satisfaction of ψ' from t. A more formal description of this framework can be found in [MNP05, MNP06]. We first present a generic automaton, the *event recorder* which was first introduced in [MNP05] for the purpose of showing that the operator $\diamondminus_{[a,b]}$ admits a deterministic timed automaton.

The automaton depicted in Figure A.2 accepts signals satisfying $\diamondminus_{[a,b]}\, \varphi$ by simply memorizing at any time instant t the value of φ in the past temporal window $[t - b, t]$. Assuming that φ is of bounded variability and cannot change more than $2m$ times in an interval of length b, the states of the automaton, $\{0, 01, \ldots, (01)^m 0\}$, correspond to the qualitative form of the value of φ in that time interval. Each clock x_i (respectively, y_i) measures the time elapsed since the i^{th} rising (respectively, falling) of φ in the temporal window. When φ first becomes true, automaton moves from 0 to 01 and resets x_1. When φ becomes false it moves to 010 while resetting y_1 and so on. When clock $y_1 > b$, the first 01-episode of φ becomes irrelevant for the satisfaction of $\diamondminus_{[a,b]}\, \varphi$ and can be forgotten. This is achieved by the "vertical" transitions which are accompanied by "shifting" the clocks values, that is, applying the operations $x_i := x_{i+1}$ and $y_i := y_{i+1}$ for all i. This allows us to use only a finite number of clocks.

The following proposition, first observed in [MN04], simplifies the construction of the automaton. It follows from the fact that if a bounded-variability signal is true at two close points, it has to be true throughout the interval between them.

Proposition A.6. *If p is a signal of $(a, 1)$-bounded variability then*

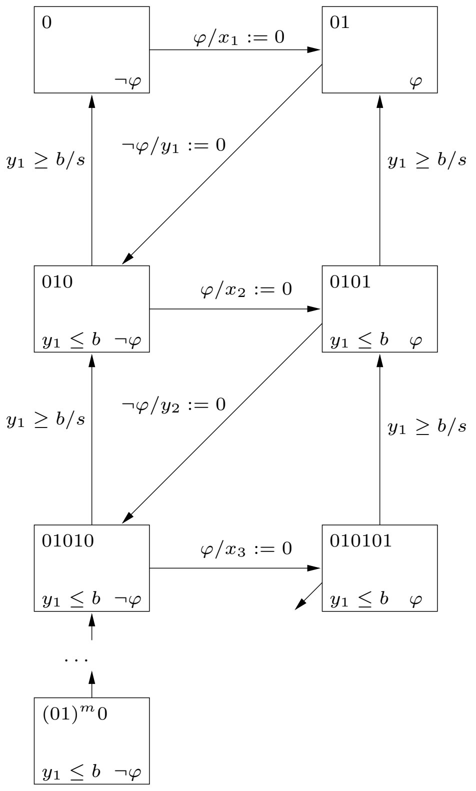


Fig. A.2. An event recorder, an automaton which has φ as input and $\diamondminus_{[a,b]}\varphi$ as output. The input labels and staying conditions are written on the bottom of each state. Transitions are decorated by the input labels of the target states and by clock resets. The clock shift operator is denoted by the symbol s. The automaton outputs 1 whenever $x_1 \geq a$.

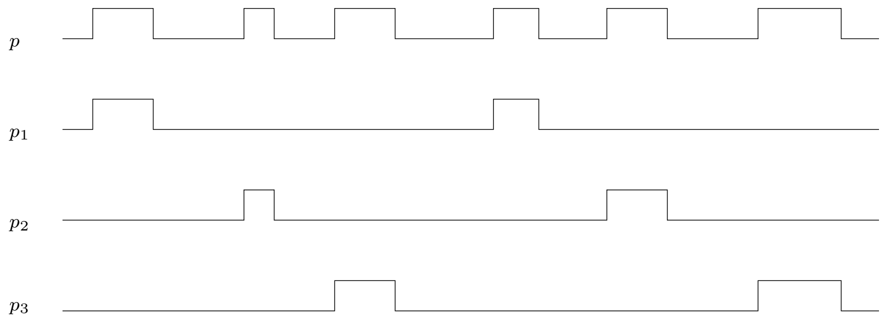


Fig. A.3. Splitting p into $p_1 \vee p_2 \vee p_3$.

- $(\xi, t) \models p\mathcal{U}_{[a,b]}q$ iff $(\xi, t) \models p \wedge \diamond_{[a,b]}(p \wedge q)$
- $(\xi, t) \models p\mathcal{P}_{[a,b]}q$ iff $(\xi, t) \models \diamondminus_b p \wedge \diamondminus_{[0,b-a]}(p \wedge q)$

Hence for a signal p satisfying this property, the automaton for $\mathcal{P}_{[a,b]}$ can be constructed from the event recorder by means of simple Boolean composition. Suppose now that p is of (a, k)-bounded variability with $k > 1$. We can decompose it into k signals $p_1, \ldots, p_k$ such that $p = p_1 \vee p_2 \cdots p_k$, $p_i \wedge p_j$ is always false for every $i \neq j$ and each p_i is of $(a, 1)$-bounded variability. This is achieved by letting p_i rise and fall only on the j^{th} rising and falling of p, where $j = i \mod k$, as is illustrated, for $k = 3$, in Figure A.3. It is not hard to see that for such p_i's we have

$$(\xi, t) \models p\mathcal{U}_{[a,b]}q \text{ iff } (\xi, t) \models \bigvee_{i=1}^{k} p_i\mathcal{U}_{[a,b]}q$$

and

$$(\xi, t) \models p\mathcal{P}_{[a,b]}q \text{ iff } (\xi, t) \models \bigvee_{i=1}^{k} p_i\mathcal{P}_{[a,b]}q.$$

The splitting of p can be done trivially using an automaton realizing a counter modulo k.

Theorem A.7 (MTL-B to Deterministic Timed Automata). *Any* MITL-B *formulae can be transformed, under bounded-variability assumptions, into equivalent deterministic timed automata.*

A.4 Application to Synthesis

A.4.1 Discrete and Dense-Time Tools

What remains to be done is to transform the automaton into a timed game automaton by distinguishing controllable and uncontrollable actions and applying the synthesis algorithm. There are currently several choices for timed synthesis tools divided into two major families depending one whether discrete or dense time tools are used.[3]

- *Discrete time*: the logic and the automata are interpreted over the time domain $\mathbb{N}$. A major advantage of this approach is that the automaton becomes finite state and can be subject to symbolic verification and synthesis using BDDs, which is very useful when the discrete state space is large. On the other hand, the sensitivity of discrete time analysis to the size of the constants is much higher and will lead to explosion when they are large. Discrete-time synthesis of scheduler for fairly-large systems has been reported in [KY03].
- *Dense time*: here we have the opposite problem, namely there is a compact symbolic representation of subsets of the clock space, but the discrete states are enumerated. Several implementations of synthesis algorithms based on [MPS95] exist. One is the tool `SynthKro` included in the standard distribution of Kronos and described in [AT02], which works by standard fixpoint computation. Another alternative, which restricts the algorithm to work only on the reachable part of the state space is the tool `FlySynth` which refines the reachability graph of the game automaton according to the time-abstract bisimulation relation [TY01] yielding a finite quotient to which *untimed* synthesis algorithms can be applied [TA99]. Finally, the tool `Uppaal-Tiga` improves upon these ideas by combining forward and backward search, resulting in the most "on-the-fly" algorithm for timed synthesis [CDF$^+$05] and probably the most effective existing tool for timed synthesis.

[3] Contrary to commonly-held beliefs, the important point of timed automata is not the density of time but the *symbolic* treatment of timing constraints using addition and inequalities rather than state enumeration.

We have conducted our first experiments in discrete time using a synthesis algorithm implemented on top of the tool TLV, while working on the implementation of an improved dense time algorithm combining ideas from [TY01] and [CDF$^+$05].

A.4.2 Example: Deriving an Arbiter

To demonstrate our approach we present a bounded-future specification of an *arbiter* module whose architectural layout is shown in Figure A.4-(a). The arbiter is expected to allocate a single resource among n clients. The clients post their *requests* for the resource on the input ports $r_1, \ldots, r_n$ and receive notification of their *grants* on the arbiter's output ports $g_1, \ldots, g_n$. The protocol of communication between each client and the arbiter follows the cyclic behavior described in Figure A.4-(b,c).

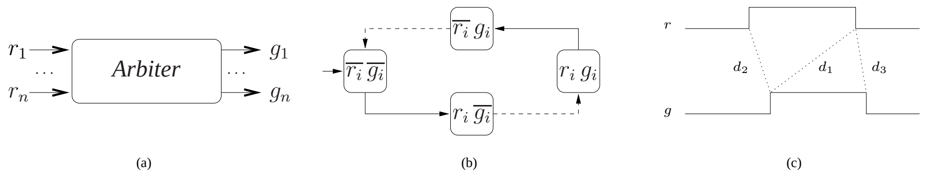


Fig. A.4. (a) The architecture of an Arbiter; (b) The communication protocol between the arbiter and client i. Uncontrollable actions of the client (environment) are drawn as solid arrows, while controllable actions which are performed by the arbiter (controller) drawn as dashed arrows; (c) A typical interaction between the arbiter and a client.

In the initial state both r_i and g_i are low (0). Then, the client acts first by setting r_i to high (1) indicating a request to access the shared resource. Next, it is the turn of the arbiter to respond by raising the *grant* signal g_i to high. Sometimes later, the client terminates and indicates its readiness to relinquish the resource by lowering r_i. The arbiter acknowledges the release of the resource by lowering down the grant signal g_i.

We structure the specification into subformulae I^E, I^C, S^E, S^C, L^E and L^C denoting, respectively, the initial condition, safety component, and (bounded) liveness components of the environment (client) and the controller (arbiter). They are given by

$$\begin{array}{ll}
I^E : \bigwedge_i \overline{r_i} & \\
I^C : \bigwedge_i \overline{g_i} & \\
S^E : \bigwedge_i r_i \implies r_i \mathcal{S}(\overline{r_i} \wedge \overline{g_i})) & \wedge \ \bigwedge_i(\overline{r_i} \implies \overline{r_i}\mathcal{B}(r_i \wedge g_i)) \\
S^C : \bigwedge_i(g_i \implies g_i \mathcal{S}(r_i \wedge \overline{g_i})) & \wedge \ \bigwedge_i(\overline{g_i} \implies \overline{g_i}\mathcal{B}(\overline{r_i} \wedge g_i)) \\
L^E : \bigwedge_i(g_i \implies \Diamond_{[0,d_1]}\overline{r_i}) & \\
L^C : \bigwedge_i(r_i \implies \Diamond_{[0,d_2]}g_i) & \wedge \ \bigwedge_i(\overline{r_i} \implies \Diamond_{[0,d_3]}\overline{g_i})
\end{array}$$

The initial-condition requirements I^E and I^C state that initially all variables are low. The safety requirements S^E and S^C ensure that the environment and arbiter conform to the protocol as described in Figure A.4-(b). In the untimed case, this is usually specified using the next-time operator $\bigcirc$ but in dense time specify these properties using the the untimed past $\mathcal{S}$ and $\mathcal{B}$ operators. Thus, the requirement $(r_i \implies r_i\mathcal{S}(\overline{r_i} \wedge \overline{g_i}))$ states that if r_i is currently high, it must have been continuously high since a preceding state in which both r_i and g_i were low. The reader can verify that the combination of the safety properties enforces the protocol.

The (bounded) liveness property $g_i \implies \Diamond_{[0,d_1]} \overline{r_i}$ requires that if g_i holds then within b time units, client C_i should release the resource by lowering r_i. The property $(r_i \implies \Diamond_{[0,d_2]} g_i)$ specifies quality of service by saying that every client gets the resource at most d_2 time after requesting it. Finally, property $\overline{r_i} \implies \Diamond_{[0,d_3]} \overline{g_i}$ requires that the arbiter senses the release of the resource within d_3 time and considers it available for further allocations. Note that the required response delays for the various properties employ different time constants. This is essential, because the specification is realizable only if d_2, the time bound on raising g, is at least $n(d_1 + d_3)$. This reflects the "worst-case" situation that all clients request the resource at about the same time, and the arbiter has to service each of them in turn, until it gets to the last one.

The various components are combined into a single MTL-B formula by transforming them to past formulae and requiring that the controller does not violate its requirements as long as the environment does not violate hers:

$$(I^E \implies I^C) \quad \wedge \quad \Box(\boxminus(\Pi(S^E) \wedge \Pi(L^E)) \implies (\Pi(S^C) \wedge \Pi(L^C))) \qquad \text{(A.2)}$$

Below we report some preliminary experiments in automatic synthesis of the arbiter. Table A.1 shows the results of applying the procedure to Equation (A.2) with $d_3 = 1$ and d_1 (the upper bound on the execution time of the client) varying between 2 and 4. The N column indicates the number of clients, the *Size* column indicate the number of BDD nodes in the symbolic representation of the transition relation of the synthesized automaton and *Time* indicates the running time (in seconds) of the synthesis procedure. As one can see, there is a natural exponential growth in N and also in d_2 as expected using discrete time.

N	d_1	d_2	Size	Time	d_1	d_2	Size	Time	d_1	d_2	Size	Time
2	2	4	466	0.00	3	5	654	0.01	4	6	946	0.02
3	2	8	1382	0.14	3	10	2432	0.34	4	12	4166	0.51
4	2	12	4323	0.63	3	15	7402	1.12	4	18	16469	2.33
5	2	16	13505	1.93	3	20	26801	4.77	4	24	50674	10.50
6	2	20	43366	8.16	3	25	84027	22.55	4	30	168944	64.38
7	2	24	138937	44.38	3	30	297524	204.56	4	36	700126	1897.56

Table A.1. Results for $d_1 = 2, 3, 4$.

A.5 Conclusions and Future Work

We have made an important step toward making synthesis a usable technology by suggesting MTL-B as a suitable formalism that can handle a variety of bounded response properties encountered in the development of real-time systems. We have provided a novel translation form real-time temporal logic to deterministic timed automata via transformation to past formulae and using the reasonable bounded-variability assumption. We have demonstrated the viability of this approach by deriving a non-trivial arbiter from specifications.

In the future we intend to focus on efficient symbolic algorithms in the spirit of [CDF$^+$05] and conduct further experiments in order to characterize the relative merits of discrete and dense-time algorithms. We also intend to apply the synthesis algorithm to more complex specifications of real-time scheduling problems.

References

[ABG$^+$00] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In *Proc. CAV'00*, pages 538–542. LNCS 1855, Springer, 2000.

[Acc04] Accelera Standard, SystemVerilog 3.1a Language Reference Manual, 2004.

[Acc08] Accelera Standard, Verilog AMS 2.3 Language Reference Manual, 2008.

[ACM02] E. Asarin, P. Caspi and O. Maler, Timed Regular Expressions, *The Journal of the ACM* **49**, 172–206, 2002.

[AD94] R. Alur and D.L. Dill, A Theory of Timed Automata, *Theoretical Computer Science* **126**, 183–235, 1994.

[ADF$^+$06] E. Asarin, T. Dang, G. Frehse, A. Girard, C. Le Guernic and O. Maler, Recent Progress in Continuous and Hybrid Reachability Analysis, *CACSD*, 2006.

[AELP99] R. Alur, K. Etessami, S. La Torre and D. Peled Parametric Temporal Logic for "Model Measuring" *ICALP'99*, 159–168, 1999.

[AFH96] R. Alur, T. Feder, and T.A. Henzinger, The Benefits of Relaxing Punctuality, *Journal of the ACM* **43**, 116–146, 1996.

[AFH99] R. Alur, L. Fix, and T.A. Henzinger, Event-Clock Automata: A Determinizable Class of Timed Automata, *Theoretical Computer Science* **211**, 253–273, 1999.

[AH92a] R. Alur and T.A. Henzinger, Logics and Models of Real-Time: A Survey, *REX Workshop, Real-time: Theory in Practice*, 74–106. LNCS 600, 1992.

[AH92b] R. Alur and T.A. Henzinger, Back to the Future: Towards a Theory of Timed Regular Languages, *FOCS'92*, 177-186, 1992.

[Alu99] R. Alur, Timed Automata, *CAV'99*, LNCS 1633, 8–22, 1999.

[AMP95] E. Asarin, O. Maler and A. Pnueli, Symbolic Controller Synthesis for Discrete and Timed Systems, *Hybrid Systems II*, 1–20, LNCS 999, 1995.

[Asa04] E. Asarin, Challenges in Timed Languages, *Bulletin of EATCS* 83, 2004.

[AT02] K. Altisen and S. Tripakis, Tools for Controller Synthesis of Timed Systems, *RT-TOOLS'02*, 2002.

[AZDT07] G. Al Sammane, M.H. Zaki, Z.J. Dong and S. Tahar, Towards Assertion Based Verification of Analog and Mixed Signal Designs Using PSL, *FDL'07*, 2007.

[BBF$^+$01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci and Ph. Schnoebelen, *Systems and Software Verification. Model-Checking Techniques and Tools*, Springer, 2001.

[BBKT04] S. Bensalem, M. Bozga, M. Krichen and S. Tripakis, Testing Conformance of Real-time Applications with Automatic Generation of Observers, *RV'04*, 2004.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L.J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, *Information and Computation* **98**, 140–170, 1992.

[BK08] C. Baier and J. P. Katoen, *Principles of Model Checking*, MIT Press, 2008.

[BL69] J.R. Büchi and L.H. Landweber, Solving Sequential Conditions by Finite-state Operators, *Trans. of the AMS* **138**, 295–311, 1969.

[CDF$^+$05] F. Cassez, A. David, E. Fleury, K.G. Larsen and D. Lime, Efficient On-the-Fly Algorithms for the Analysis of Timed Games, *CONCUR'05*, 66–80, 2005.

[CE81] E. M. Clarke and E. A. Emerson, Design and Synthesis of Synchronization Skeletons Using Branshing Time Temporal Logic, In *Logic of Programs*, **131**, 1981.

[CGH94] E. M. Clarke, O. Grumberg and K. Hamaguchi, Another look at LTL Model Checking, *CAV'94*, 415–427, LNCS 818, 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*. The MIT Press, 1999.

[Chu63] A. Church, Logic, Arithmetic and Automata, in *Proc. of the Int. Cong. of Mathematicians 1962*, 23–35, 1963.

[CKS81] A. K. Chandra, D. C. Kozen and J. J.Stockemeyer, Alternation, *Journal of ACM*, **28(1)**, 114–133, 1981.

[CRST06] A. Cimatti, M. Roveri, S. Semprini and S.Tonetta, From PSL to NBA: a Modular Symbolic Encoding, *FMCAD*, 125–133, 2006.

[Dam08] M. Damler, *What is the single greatest need for enhanced EDA solutions in AMS design?*, Analog Insights: Analog/Mixed Signal Verification Blog, Synopsys OpenCommunity, 2008. http://synopsysoc.org/analoginsights/?p=71

[DC05] T.R. Dastidar and P.P Chakrabarti, Verification System for Transient Response of Analog Circuits Using Model Checking, *VLSID'05*, 195–200, 2005.

[Dil89] D. Dill, Timing Assumptions and Verification of Finite-State Concurrent Systems, In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, 197–212, 1989.

[Dru00] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proc. SPIN'00*, pages 323–330. LNCS 1885, Springer, 2000.

[DT04] D. D'Souza and N. Tabareau, On Timed Automata with Input-determined Guards, *FORMATS/FTRTFT'04*, 68-83, LNCS 3253, 2004

[DY96] C. Daws and S. Yovine, Reducing the Number of Clock Variables of Timed Automata, *RTSS'96*, 73–81, 1996.

[EFH$^+$03] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac and D. van Campenhout, Reasoning with Temporal Logic on Truncated Paths, CAV'03, 27–39, LNCS 2725, 2003.

[EFH05] C. Eisner, D. Fisman and J. Havlicek A Topological Characterization of Weakness, In *PODC'05*, 2005.

[FGP06] G. Fainekos, A. Girard and G. Pappas Temporal Logic Verification Using Simulation In *Proc. FORMATS'06*, pages 171–186. LNCS 4202, Springer, 2006.

[Gei02] M. C. W. Geilen, *Formal Techniques for Verification of Complex Real-time Systems,* PhD. Thesis. Eindhoven University of Technology, 2002.

[GO01] P. Gastin and D. Oddoux, Fast LTL to Büchi Automata Translation, *CAV'01*, 53–65, LNCS 2102, 2001.

[GPVW95] R. Gerth, D.A. Peled, M.Y. Vardi and P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, *PSTV*, 3–18, 1995.

[Hen96] T. .A. Henzinger, The Theory of Hybrid Automata, *LICS'96*, **170**, 278–292, 1996.

[Hen98] T.A. Henzinger, It's about Time: Real-time Logics Reviewed, *CONCUR'98*, 439–454, LNCS 1466, 1998.

[HFE04] J. Havlicek, D. Fisman and C. Eisner, Basic results on the semantics of Accellera PSL 1.1 foundation language, *Technical Report 2004.02*, Accelera, 2004.

[HNSY94] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, Symbolic Model-checking for Real-time Systems, *Information and Computation* **111**, 193–244, 1994.

[HR01] K. Havelund and G. Rosu. Java PathExplorer - a Runtime Verification Tool. In *Proc. ISAIRAS'01*, 2001.

[HR02] K. Havelund and G. Rosu, Synthesizing Monitors for Safety Properties, *TACAS'02*, 342–356, LNCS 2280, 2002.

[HR04a] Y. Hirshfeld and A. Rabinovich, Logics for Real Time: Decidability and Complexity, *Fundamenta Informaticae* **62**, 1–28, 2004.

[HR04b] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*, Cambridge University Press, 2004.

[HRS98] T. A. Henzinger, J. F. Raskin and P. Y. Schobbens, The Regular Real-Time Languages, *ICALP'98*, 580–591, 1998.

[JKN08] K. D. Jones, V. Konrad and D. Nickovic, Analog Property Checkers: A DDR2 Case Study, *FAC'08*, 2008.

[Iee01] IEEE Standard Verilog Hardware Description Language IEEE Std. 1363-2001, 2001.

[Jed06] JEDEC Standard, JESD79-2C DDR2 SRAM Specification

[JHP$^+$07] A. Jesser, S. Lämmermann, A. Pacholik, R. Weiss, J. Ruf, W. Fengler, L. Hedrich, T. Kropf and W. Rosenstiel, Analog Simulation Meets Digital Verification - A Formal Assertion Approach for Mixed-Signal Verification SASIMI'07, 507–514, 2007.

[Kam68] H. Kamp, *Tense Logic and the Theory of Linear Order*, PhD Thesis, University of California, Los Angeles, 1968.

[KC06] C. Kossentini and P. Caspi, Approximation, Sampling and Voting in Hybrid Computing Systems, *HSCC*, 2006.

[KLS$^+$02] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, Checking, and Steering of Real-time Systems. In *Proc. RV'02*. ENTCS 70(4), 2002.

[KMP94] Y. Kesten, Z. Manna and A. Pnueli, Temporal Verification of Simulation and Refinement, *A Decade of Concurrency*, LNCS 803, 276–346, 1994.

[Koy90] R. Koymans, Specifying Real-time Properties with with Metric Temporal Logic, *Real-time Systems* **2**, 255–299, 1990.

[KP05] Y. Kesten and A. Pnueli, A Compositional Approach to CTL* Verification, *Theoretical Computer Science* **331**, 397–428, 2005.

[KPA03] K. J. Kristoffersen, C. Pedersen and H. R. Andersen, Runtime Verification of Timed LTL using Disjunctive Normalized Equation Systems, *RV'03*, ENTCS 89(2), 2003.

[KPR98] Y. Kesten, A. Pnueli and L. Raviv, Algorithmic Verification of Linear Temporal Logic Specifications, *Int. Colloq. Aut. Lang. Prog* **1443**, 1–16, 1998.

[KV01] O. Kupferman and M. Vardi, On Bounded Specifications, in *LPAR'01*, **2250**, 24–38, 2001.

[KVR83] R. Koymans, R. Vytopil and W. P. de Roever, Real-Time Programming and Asynchronous Message-Passing, *Symp. on Principles of Distributed Computing*, 187–197, 1983.

[KY03] Ch. Kloukinas and S. Yovine, Synthesis of Safe, QoS Extendible, Application Specific Schedulers for Heterogeneous Real-Time Systems, *ECRTS'03*, 287–294, 2003.

[KZ04] K. S. Kundart and O. Zilke, *The Designer's Guide to Verilog-AMS*, Kluwer Academic Publishers, 2004.

[LMS02] F. Laroussine, N. Markey and P. Schnoebelen, Temporal Logic with Forgettable Past, *LICS'02*, 383–392, 2002.

[LPZ85] O. Lichtenstein, A. Pnueli and L. Zucks, The Glory of the Past, *Conf. on Logic of Programs*, 192–218, 1985.

[Mal06] O. Maler, Analog Circuit Verification: a State of an Art *ENTCS* 153, 3-7, 2006.

[Mal07] O. Maler, On Optimal and Reasonable Control in the Presence of Adversaries, *Annual Reviews in Control*, 2007.

[Mau08] A. Mauskar, *Analog tools must catch up*, EE Times, 2008, http://www.eetimes.com/news/design/showArticle.jhtml?articleID=206905585

[MH84] S. Miyano and T. Hayashi, Alternating Finite Automata on ω-Words, *TCS* 32, 321-330, 1984.

[Mic84] M. Michel, Algèbre de Machines et Logique Temporelle, *STACS'84*, 1984.

[Mic85] M. Michel, Composition of Temporal Operators, *Logique et Analyse*, 110-111, 137-152, 1985.

[MMP92] O. Maler, Z. Manna and A. Pnueli, From Timed to Hybrid Systems *Real-Time: Theory in Practice*, 447-484, LNCS 600, 1992.

[MN04] O. Maler and D. Nickovic Monitoring Temporal Properties of Continuous Signal *FORMATS/FTRTFT'04*, 152–166, 2004.

[MNP05] O. Maler, D. Nickovic and A. Pnueli Real-Time Temporal Logic: Past, Present, Future, *FORMATS'05*, 2–16, 2005.

[MNP06] O. Maler, D. Nickovic and A. Pnueli, From MITL to Timed Automata, *FORMATS'06*, 2006.

[MNP07a] O. Maler, D. Nickovic and A. Pnueli, On Synthesizing Controllers from Bounded-Response Properties, *CAV'07*, 2007.

[MNP07b] O. Maler, D. Nickovic and A. Pnueli, Checking Temporal Properties of Discrete, Timed and Continuous Behaviors, *Pillars of Computer Science'07*, 2007.

[Mos99] P.J. Mosterman, An Overview of Hybrid Simulation Phenomena and their Support by Simulation Packages, *HSCC'99*, 165-177, LNCS 1569, 1999.

[MP90] O. Maler and A. Pnueli, Tight Bounds on the Complexity of Cascaded Decomposition of Automata, *FOCS'90*, 672–682, 1990.

[MP91] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Specification*, Springer, 1991.

[MP95a] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[MP95b] O. Maler and A. Pnueli, Timing Analysis of Asynchronous Circuits using Timed Automata, *CHARME'95*, 189-205, LNCS 987, Springer, 1995.

[MP04] O. Maler and A. Pnueli, On Recognizable Timed Languages, *FOSSACS'04*, 348–362, LNCS 2987, 2004.

[MPS95] O. Maler, A. Pnueli and J. Sifakis, On the Synthesis of Discrete Controllers for Timed Systems, *STACS'95*, 229–242, LNCS 900, 1995.

[MR05] N. Markey and J. F. Raskin, *Model Checking Restricted Sets of Timed Paths*, Theoretical Computer Science, **358**, 2005.

[MS03] N. Markey and Ph. Schnoebelen, Model Checking a Path, *CONCUR'03*, 251–265, LNCS 2761, 2003.

[Nan08] NanoSim: Memory and Mixed-Signal Verification, Datasheet, Synopsys, 2008.

[ND07a] T. Nahhal and T. Dang, Test Coverage for Continuous and Hybrid Systems *CAV'07*, 449–462, 2007.

[ND07b] T. Nahhal and T. Dang, Guided Randomized Simulation *HSCC'07*, 731–735, 2007.

[NM07] D. Nickovic and O. Maler, AMT: A Property-Based Monitorting Tool for Analog Systems *FORMATS'07*, 304–319, 2007

[NP86] M. Nivat and D.Perrin, Ensembles Reconnaissables de Mots Bi-infinis, *Canadian J. of Mathematics*, **38**, 513–537, 1986.

[OW05] J. Ouaknine and J. Worrell, On the Decidability of Metric Temporal Logic, *LICS'05*, 188–197, 2005.

[Pnu77] A. Pnueli, The temporal Logic of Programs, *Symposium on Foundations of Computer Science*, 46–57, 1977.

[PPS06] N. Piterman, A. Pnueli and Y. Sa'ar, Synthesis of Reactive(1) Designs, *VMCAI'06*, 364–380, 2006.

[PP06] N. Piterman and A. Pnueli, Faster Solutions of Rabin and Streett Games, *LICS'06*, 275–284, 2006.

[PR89] A. Pnueli and R. Rosner, On the Synthesis of a Reactive Module, *POPL'89*, 179–190, 1989.

[PZ06a] A. Pnueli and A. Zaks, PSL Model Checking and Run-time Verification via Testers, *International Symposium on Formal Methods*, **4085**, 573–585, 2006.

[PZ06b] A. Pnueli and A. Zaks, On the Merits of Temporal Testers, *25 Years of Model Checking*, 2006.

[QS82] J. P. Queille and J. Sifakis, Specification and Verification of Concurrent Systems in CESAR, In *Proceedings of the 5th International Symposium on Programming*, 337–350, 1982.

[Rey03] M. Reynolds, The Complexity of the Temporal Logic with Until over General Linear Time, *Journal of Computer and System Sciences*, **66**, 393–426, 2003.

[RS97] J.-F. Raskin and P.Y. Schobbens, State-clock logic: a decidable real-time logic, In *Hybrid and Real Systems*, 33–47, 1997.

[RSH98] J.-F. Raskin, P.Y. Schobbens and T.A. Henzinger, Axioms for Real-Time Logics, *Concur'98*,

[RW89] P.J. Ramadge and W.M. Wonham, The Control of Discrete Event Systems, *Proc. of the IEEE* **77**, 81–98, 1989.

[SB00] F. Somenzi and R. Bloem, Efficient Büchi automata from LTL formulae, *CAV'00*, 248–263, LNCS 1855, 2000.

[Sei08] N. Seiden, *Why we need an analog design flow that's like digital now*, EDA DesignLine, 2008. http://www.edadesignline.com/howto/205210265

[Sub07] R. Subramanian, *Verification Challenges Facing Analog, RF Designers*, Nikkei Electronics Asia, 2007. http://techon.nikkeibp.co.jp/article/HONSHI/20070831/138599/

[Syn04] *CosmosScope Reference Manual*, Synopsis, 2004.

[TA99] S. Tripakis and K. Altisen, On-the-Fly Controller Synthesis for Discrete and Timed Systems, *FM'99*, 1999.

[TB73] B.A. Trakhtenbrot and Y.M. Barzdin, *Finite Automata: Behavior and Synthesis*, North-Holland, Amsterdam, 1973.

[TR04] P. Thati and G. Rosu, Monitoring Algorithms for Metric Temporal Logic Specifications, *RV'04*, 2004.

[Tra04] B.A. Trakhtenbrot, Understanding Basic Automata Theory in the Continuous Time Setting, *Fundamenta Informatica* 62, 69-121,2004

[Tri02] S. Tripakis, Fault Diagnosis for Timed Automata, *FTRTFT'02*, 205–224, LNCS 2469, 2002.

[TY01] S. Tripakis and S. Yovine, Analysis of Timed Systems using Time-abstracting Bisimulations, *Formal Methods in System Design* **18**, 25–68, 2001.

[TYB05] S. Tripakis, S. Yovine and A. Bouajjani, Checking Timed Büchi Automata Emptiness Efficiently, *Formal Methods in System Design* **26**, 267-292, 2005.

[Var95] M.Y. Vardi, Alternating Automata and Program Verification, *Computer Science Today*, 471-485, LNCS 1000, 1995.

[VW86] M.Y. Vardi and P. Wolper, An Automata-theoretic Approach to Automatic Program Verification, *LICS'86*, 322–331, 1986.

[Yov97] S. Yovine, Kronos: A Verification Tool for Real-time Systems, *International Journal of Software Tools for Technology Transfer* **1** 123–133, 1997.

[Zuc86] L. Zuck, *Past Temporal Logic*, PhD. Thesis, Weizmann Institute, 1986.