Modeling, Verification and Testing using Timed and Hybrid Automata

Stavros Tripakis and Thao Dang

September 12, 2008

Contents

Chapter 1

Modeling, Verification and Testing using Timed and Hybrid Automata

1.1 Introduction

Models have been used for a long time to build complex systems, in virtually every engineering field. This is because they provide invaluable help in making important design decisions, before the system is implemented. Recently, the term *model-based design* has been introduced to emphasize the use of models and place them in the center of the development process, especially for software-intensive systems. Traditionally, the fact that software is immaterial (contrary, say, to bridges or cars or hardware), has resulted in a software development process that largely blurs the line between design and implementation: a model of the software is the software itself, which is also the implementation! It is "cheap" to write software and test it, or so people used to believe. It is now becoming more and more clear that the costs for software development, testing and maintenance are non-negligible, in fact, they often outweigh the costs of the rest of the system.

As a result of this and other factors, a more rigorous software development process based on formal, high-level models, is becoming widespread, especially in the *embedded software* domain. The term "embedded software" may generally include any type of software that runs on an embedded system. Embedded systems are computer-controlled systems that strongly interact with a physical environment, for example, *x-by-wire* systems for car control, cell phones, multimedia devices, medical devices, defense and aerospace, public transportation, energy and chemical plants, etc.

In all these systems, timing and other physical characteristics of the environment are essential for system correctness as well as for performance. For instance, in an engine-control system it is critical to ignite the engine at very precise moments in time (in the order of 1 millisecond). Also, the control logic depends on a number of continuously evolving variables that have to do with the combustion in the engine, exhaust, and so on. In order to capture such timing and physical constraints, models of *timed automata* and *hybrid automata* have been developed in the early '90s [9, 5]. Since then, these models have been studied extensively and today there are a number of sophisticated analysis and synthesis methods and tools available for such models. We will discuss some of these methods in this chapter.

Models, in general, play different roles and are used for different tasks, at different phases of the design process. For instance, sometimes a model captures a system that is already built to some extent, while at other times a model serves as a specification that the final system must conform to. In the rest of this chapter, we consider the following tasks in the context of timed or hybrid automata models:

- *Modeling*: We discuss timed and hybrid automata in Section 1.2. Modeling is of course a task by itself, and probably the most crucial one, since it is a creative and to a large extent non-automatable task.
- *Exhaustive verification*: We use the term exhaustive verification to denote the task of proving that a given model of a system satisfies a given property (also expressed in some modeling language). This task is exhaustive in the sense that it is conclusive: either the proof succeeds or a counter-example is found that demonstrates that the system fails to satisfy the property. We focus on automatic ("push-button") exhaustive verification, also called *model checking*. We discuss exhaustive verification in Section 1.3.
- *Partial verification*: Fundamental issues such as undecidability (in the case of hybrid

automata) or state-explosion (in the case of both timed and hybrid automata) place limits on exhaustive verification. In Section 1.4 we discuss an alternative, namely partial verification, which aims to check a given model as much as possible, given time and resource constraints. This is done using mainly simulation-based methods.

- *Testing*: It is important to test correctness of a system even after it is built. Testing mainly serves this purpose. Since designing good and correct test cases is itself a time-consuming and error-prone process, one of the main challenges in testing is automatic test generation from a formal specification. We discuss testing in general in Section 1.5, and test generation from timed and hybrid automata models in Sections 1.6 and 1.7, respectively.

Obviously, the topics covered in this chapter are wide and deep, and we can only offer an overview. We attempt an intuitive presentation and omit most of the technical details. Those can be found in the referenced papers. We also restrict ourselves to the topics mentioned above and omit many others. For example, we do not discuss discrete-time/state models, theorem proving, controller synthesis, implementability and code generation, as well as other interesting topics. Finally, excellent surveys exist for some of the topics covered in this chapter (e.g., see Alur's survey on timed automata [3] and an overview of hybrid systems in the book [107]), thus we prefer to devote more of our discussion to topics that are more recent and perhaps have been less widely exposed, such as testing and partial verification.

1.2 Modeling with timed and hybrid automata

Timed and hybrid automata models are extensions of finite automata with variables that evolve continuously in time. Timed automata are a subclass (i.e., special case) of hybrid automata. In timed automata all variables evolve with rate 1: that is, these variables measure time itself. Hybrid automata are more general, with variables that can in principle obey any type of continuous dynamics, usually expressed by some type of differential equations.

These models were introduced in order to meet the desire to blend the "discrete" world of computers with the "continuous" physical world. Classical models from computer science (e.g., finite-state automata) provide means for reasoning about discrete systems only. Classical models from engineering (e.g., differential equations) provide means for reasoning mostly about continuous systems. Timed and hybrid automata are attempts to bridge the two worlds. Although timed automata are a special class of hybrid automata, their study as a separate model is justified by the fact that many problems that are very difficult or impossible to solve (i.e., undecidable) for hybrid automata, are easier or solvable for timed automata.

1.2.1 Timed automata

Timed automata [9] extend finite automata by adding variables that are able to measure real-time: these variables are called *clocks*. Standard finite-state automata are able to specify that a certain set of events occurs in a specific order, however, they do not typically specify how much time has elapsed between two successive events. Figure 1.1(a) shows an example of a finite automaton that specifies the order between four events a, b, c, d: event a precedes b, which precedes c, which precedes d. This automaton has five states numbered 0 to 4, and four transitions labeled by the four events. State 0 is the initial state.

$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[r, "a"] & 1 \arrow[r, "b"] & 2 \arrow[r, "c"] & 3 \arrow[r, "d"] & 4
\end{tikzcd}$$

(a) A finite state automaton

$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[r, "a", "x := 0"'] & 1 \arrow[r, "b", "y := 0"'] & 2 \arrow[r, "c", "y \geq 2"'] & 3 \arrow[r, "d", "x \leq 5"'] & 4
\end{tikzcd}$$

(b) A timed automaton

Figure 1.1: A finite-state automaton and a timed automaton.

Figure 1.1(b) shows an example of a timed automaton (TA). It is very similar to the "untimed" version, but its transitions are annotated with additional information, referring to clocks x and y. This TA specifies, in addition to the order a, b, c, d between events, two timing constraints: (1) the time that may elapse between a and d is at most 5 time units; (2) the time that may elapse between b and c is at least 2 time units. In other words, we can view the semantics of this automaton as a set of *all possible* sequences of occurrences of events in time (*timed sequences*) that satisfy timing constraints (1) and (2) in addition to the correct order a, b, c, d. This is illustrated in Figure 1.2.

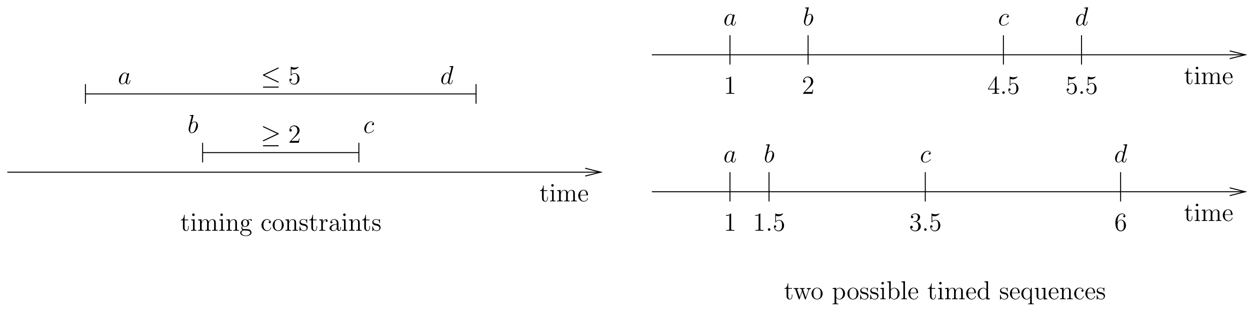

Figure 1.2: Behaviors of timed automaton of Figure 1.1.

We can also look at semantics of this automaton in an *operational* way. This is illustrated in Figure 1.3. The automaton starts at state 0 and spends a certain amount of time t_0 there. During this time the value of each clock of the automaton increases by the amount of time that has elapsed, that is, t_0 in this case. The automaton then "jumps" to state 1: event a occurs in this jump, which is instantaneous. The automaton proceeds in the same pattern: it spends some time t_1 in state 1, then jumps to state 2, and so on. The automaton alternates between these *timed* and *discrete transitions*. During a discrete transition, some clocks may be reset to zero, denoted by $x := 0$, $y := 0$, etc. Discrete transitions may have *guards*, that is, conditions such as $y \geq 2$ or $x \leq 5$, that must be satisfied in order for the transition to be possible.

The operational view reveals that knowing what state the automaton occupies at a given point in time (numbered 0, 1, 2, ..., in the above examples) is not enough to predict its future behavior: one must also know the values of its clocks (e.g., in order to check whether the guards are satisfied). This is why we must distinguish between the "discrete" state of the automaton (also called sometimes *control state* or *location*) and its "full" state which includes the clock values (sometimes called *configuration* and sometimes simply the *state*).

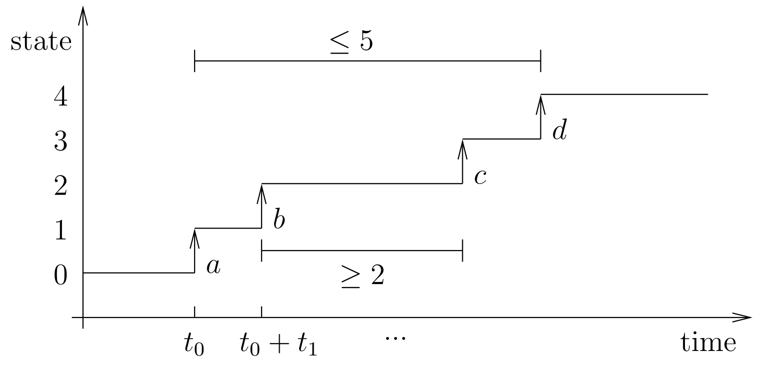

Figure 1.3: Operational semantics of timed automaton of Figure 1.1.

We will use state vs. configuration to distinguish the two.

Notice that the unit of time, although it is assumed to be the same for all the clocks, is not explicit in a timed automaton model. This is often an advantage, especially when we are only interested in the relative magnitude of timing constraints and not their absolute value. In the Alur-Dill model of timed automata, time is *dense*: delays can be taken to be positive real or rational numbers. This model is strictly more expressive than a discrete-time model, where delays are integer multiples of some given quantum of time. For instance, the dense-time model can express that two events a and b can occur arbitrarily close to each other, but not at the same time, using a "strict" constraint of the form $x > 0$. Whether to opt for a dense or discrete TA model depends on the application at hand. Considerations need to take into account not only modeling requirements, but also complexity of the algorithmic analysis, such as model checking. Dense-time model-checking is more expensive than discrete-time model-checking in theory, and often[1] in practice as well [24]. The discrete- vs. dense-time debate is a non-trivial topic. In-depth studies can be found in [48, 19, 81].

The basic model of timed automata described above can be extended in various ways (one is by adding more powerful continuous dynamics, which leads to hybrid automata described in the next section). Discrete variables can be added to the model, with basic types such as booleans or integers, but also more complex types such as records, queues, and so on. These

[1]But not always, as sometimes *symbolic* dense-time model-checking tools can represent timing constraints more effectively than *enumerative* discrete-time methods. For instance, if a guard involves large constants such as $x \leq 10^6$ then a brute-force discrete-time enumeration method with time step 1 may need to represent 10^6 distinct states while a symbolic method can represent an infinite set of states with the symbolic constraint $x \leq 10^6$.

extensions are very handy when modeling other than very simple examples. As long as the domain of such variables can be restricted to be finite, these extensions do not add to the expressive power of the model, since they can be encoded in the state of the automaton. Note that things become more complicated when attempting to relate such variables to clocks, for instance, resetting a clock x to the value of a variable i, as in $x := i$, or comparing a clock to a variable in a guard, as in $x \leq i$. Some of these extensions can be handled, but others may strictly increase the power of the model, leading even to undecidability! Again this is a non-trivial topic, and the reader is referred to [3, 23].

Another interesting extension is modeling *urgency*. To motivate this concept, consider the example shown in Figure 1.1(b). If the automaton stays in state 2 more than 5 time units, then it can no longer reach state 4. We may want to disallow this behavior, thus, model the assumption that state 4 *will* be reached. We can do this by adding acceptance conditions to the automaton (e.g., making state 4 accepting and the others non-accepting). But a more convenient way is to state this using clock constraints. For instance, we can impose the constraint $x \leq 5$ at *states* 1, 2, and 3, expressing the fact that the amount of time spent in those states must be such that this constraint is not violated. This is one way of modeling urgency, and these state-associated clock constraints are called *invariants* [49]. Another, more elaborate way is to use *deadlines*, associated with transitions [93, 18].

Even for relatively simple systems, modeling the entire system as a single automaton can be very tedious. A solution is to build a model by composing other models. In the case of timed automata, different variants of compositions have been proposed, where the components can communicate through *rendez-vous* type of action synchronization, FIFO queues, shared variables, etc. A common assumption in most of these composition frameworks is that the clocks of all automata measure exactly the same time, in other words, that they are perfectly synchronized. This is obviously unrealistic when these clocks model real clocks. Unfortunately, modeling phenomena such as clock drift explicitly (e.g., by defining the rate of a clock x to be $1 \pm \epsilon$ for a fixed $\epsilon > 0$) yields an undecidable model, in general. As an alternative, some researchers studied an asymptotic version of the problem where ϵ can be arbitrarily small [88, 109]. This allows to regain decidability while providing a more "robust" semantics. The issue of robustness is especially important when the timed automaton model is to be implemented, for instance, as an embedded controller. However, the problem can also be tackled with standard semantics, using appropriate modeling techniques [2].

Regarding applications, it is fair to say that timed automata have not found as widespread usage as standard, "untimed" models. This is not surprising, given the fact that TA are a more specialized model in the sense that often a discrete-time model is sufficient and this can be captured in a more standard language (e.g., see [24]). Moreover, TA are more expensive to analyze than "untimed" models. Still, timed automata are appealing because of their "declarative" style of specifying timing constraints, that is suitable for capturing high-level models and specifications.[2] TA have been used to model small- to medium-size systems, such as communication protocols, digital circuits, real-time scheduling systems, robotic controllers, and so on. Up-to-date lists of case studies can be found at the websites of timed-automata model-checking tools such as Kronos[3] and Uppaal[4] as well as in the publications of the authors of these tools.

1.2.2 Hybrid automata

Hybrid automata [5] can be seen as an extension of timed automata with more general dynamics. A clock c is a continuous variable with time derivative equal to 1, that is $\dot{c}(t) = 1$. In a hybrid automaton, the continuous variables x can evolve according to some more general differential equations, for example $\dot{x} = f(x)$. This allows hybrid automata to capture not only the evolution of time but also the evolution of a wide range of physical entities. The discrete dynamics of hybrid automata can also be more complex and described with more general constraints.

In the following, we present a commonly used version of hybrid automata. Different forms of constraints result in different variants of this model. A hybrid automaton $\mathcal{A}$ consists of a finite set Q of discrete states and a set of n continuous variables evolving in a continuous state space $\mathcal{X} \subseteq \mathbb{R}^n$. In each discrete state $q \in Q$, the evolution of the continuous variables

[2]It is perhaps for this reason that some of the concepts in the timed automata model have found their way into the MARTE (Modeling and Analysis of Real-time and Embedded Systems) profile for UML2. See http://www.omg.org/technology/documents/profile_catalog.htm.

[3]See http://www-verimag.imag.fr/TEMPORISE/kronos and http://www-verimag.imag.fr/ tripakis/openkronos.html.

[4]See http://www.uppaal.com.

are governed by a differential equation: $\dot{x}(t) = f_q(x(t), u(t))$ where $u(\cdot) \in \mathcal{U}_q$ is an admissible input function of the form $u : \mathbb{R}^+ \to U_q \subset \mathbb{R}^m$. This input can be used to model some external disturbance or under-specified control. A thermostat is a typical system that can be described by a hybrid automaton. The room temperature x evolves according to $\dot{x}(t) = -x(t) + u(t) + v(t)$ when the thermostat is on, and according to $\dot{x}(t) = -x(t) + v(t)$ when the thermostat is off. The input v is a disturbance input modelling the influence of the outside temperature and u is a control input modelling the heating power.

The invariant of a discrete state q is defined as a subset $\mathcal{I}_q$ of $\mathcal{X}$. The system can stay at q if $x \in \mathcal{I}_q$. The conditions for switching between discrete states are specified by a set of guards such that for each discrete transition $e = (q, q')$, the guard set $\mathcal{G}_e \subseteq \mathcal{I}_q$. Each transition $e = (q, q')$ is additionally associated with a reset map $\mathcal{R}_e : \mathcal{G}_e \to 2^{\mathcal{I}_{q'}}$ that defines how the continuous variables x may change when $\mathcal{A}$ switches from q to q'. For example, a non-deterministic linear reset map can be defined as follows: for each $x \in \mathcal{G}_e$ the new continuous state $x' = Rx + \varepsilon$ where R is a $n \times n$ matrix and $\varepsilon \in P \subseteq \mathcal{X}$. The set P models the uncertainty of this reset map.

The functions f_q are often assumed to be Lipschitz continuous and the admissible input functions $u(\cdot)$ piecewise continuous. This ensures the existence and uniqueness of solutions of the differential equations of each discrete state. However, because of complicated interactions between the continuous and discrete dynamics, further conditions are needed to guarantee the existence of a global solution of a hybrid automaton [74].

A state (q, x) of $\mathcal{A}$ can change in two ways as follows: (1) by a *continuous evolution*, the continuous state x evolves according to the dynamics f_q while the discrete state q remains constant; (2) by a *discrete evolution*, x satisfies the guard of an outgoing transition, the system changes discrete state by taking this transition and possibly changing the values of x according to the associated reset map.

Figure 1.4 sketches a hybrid automaton with two discrete states q_1 and q_2 and the continuous state space $\mathcal{X}$ is a 2-dimensional bounding rectangle. The invariant $\mathcal{I}_1$ of q_1 is the upper part of the rectangle limited by the bold line, and the invariant $\mathcal{I}_2$ of q_2 is the rectangle limited by the dashed line. The figure also shows a trajectory starting from a hybrid state

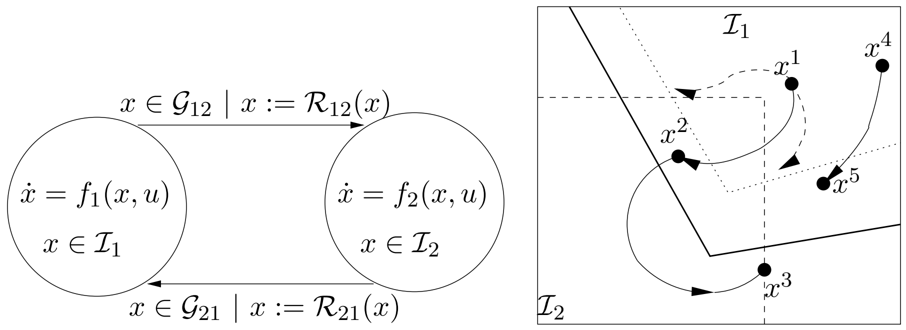

Figure 1.4: A hybrid automaton and its trajectories.

(q, x^1), which first follows the dynamics f_1 under some input $u(\cdot)$. Under different inputs, the system generates different trajectories[5] (such as, the dotted curves in the figure). The infiniteness of the input space results in an infinite number of trajectories starting from the same state, which forms a dense set, often called a *reach tube* [66].

When this trajectory reaches the guard set $\mathcal{G}_{12}$ (which is the band between the bold and the dotted line), the transition from q_1 to q_2 is enabled. At this point, the invariant condition of q_1 is still satisfied and the system can either switch to q_2 or continue with the dynamics of q_1. The former is the case of this example. The system "decides" to switch to q_2 when it reaches point x^2. The reset $\mathcal{R}_{12}$ is the identity function and thus the trajectory starts following the dynamics f_2 from the same point x^2. When the trajectory reaches a point x^3 in the guard $\mathcal{G}_{21}$ (which is the dashed boundary of $\mathcal{I}_2$), it switches back to q_1 and the application of the reset $\mathcal{R}_{21}$ to x^3 results in a new state x^4, from which the system evolves again under the dynamics f_1.

As illustrated with this example, it is important to note that this model allows to capture *non-determinism* in both continuous and discrete dynamics. This non-determinism is useful for describing disturbances from the environment as well as for taking into account imprecision in modeling and implementation.

[5]We use the term "trajectory" instead of "execution" to give a geometric intuition.

1.3 Exhaustive verification

We use the term *exhaustive verification* to signify an automated proof that a certain model satisfies a certain property. This problem is also called *model checking*. Since what we want is a proof, if we succeed in obtaining it, we can be certain that the model indeed satisfies the property. We contrast this to *partial verification* methods that are discussed in the following section.

In this section, we review exhaustive verification for timed and hybrid automata. The problem is decidable (but expensive!) for timed automata and undecidable for hybrid automata in general. In what follows we survey basic methods to tackle the problem for the two models.

1.3.1 Model checking for timed automata

The model checking problem for timed automata can be stated as: given a timed automaton (or a set of communicating timed automata) A, and given a property P, check whether A satisfies P. We will briefly review in this section methods to answer this question for different types of properties P. This is an extensively studied topic for which tutorials and surveys are already available (for instance, see [3]). For this reason, we will only sketch the basic ideas and refer to the literature for an in-depth study.

The simplest type of property P is *reachability*: we want to know whether a given state (or configuration) s of the automaton is *reachable*, that is, whether there exists an execution starting at some initial state (or set of possible initial states) and reaching s. Consider again the TA shown in Figure 1.1(b). Is state 4 reachable? It is, and Figure 1.3 presents an example execution that reaches state 4. Suppose, however, that we replaced the condition $y \geq 2$ in this automaton by $y > 5$. In that case, state 4 would become unreachable as can be verified by the reader.

Reachability is not only the simplest, but also the most useful type of property. *Safety* properties (those that state that the system has no "bad" behaviors, informally speaking) can be reduced to reachability with the help of a *monitor*. A monitor for a given property is a "passive" component that observes the behavior of the system and checks whether the property is satisfied. If the property is violated then the monitor enters a designated "bad" state. Checking whether the system satisfies the property can then be reduced to checking whether the "bad" state of the monitor is reachable in the composition of the system and the monitor.

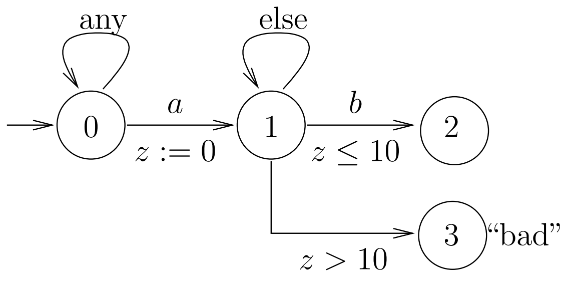

Figure 1.5: A timed-automaton monitor for checking a bounded-response property.

An example illustrating monitors is shown in Figure 1.5. The figure shows a monitor for the property "a is always followed by b within at most 10 time units". Notice that the monitor tries to capture the violation of the property, that is, its negation. In particular, the monitor synchronizes with the system on common labels. The label "any" in the self-loop transition at state 0 of the monitor is a short-hand for "any label of the system": notice that this includes the label a, that is, the monitor is non-deterministic. This is essential, because the monitor should check that the property holds on *any* execution and *every* occurrence of a. After picking an a at random, the monitor keeps track of the time using its (local) clock z. If b is observed no later than 10 time units after a, the monitor moves to the "pass" state 2. Otherwise, the monitor can move to the "bad" state 3: if the monitor is able to reach this state, then the property is violated. The "else" label stands in this case for "any label except b".

How to check reachability for timed automata? In the case of a discrete-time semantics, the problem can be reduced to a problem of checking reachability for a discrete state-transition system: configurations can be seen as vectors of non-negative integers, where the first element of the vector corresponds to the state and the rest to the values of the clocks.[6]

[6]We can always normalize the constants appearing in the timing constraints of the automaton so that the time quantum is

Moreover, the system can be *abstracted* into a finite-state system by ceasing to increment clocks whose value exceeds a certain constant c_{max}: this is the greatest constant with which a clock is compared in the automaton. For instance, in the example of Figure 1.1(b), $c_{max} = 5$. When a clock's value exceeds c_{max} we only need to "remember" this fact, and not the precise value of the clock, since this does not influence the satisfaction of a timing constraint. With this observation, one is left with the task of verifying exhaustively a finite-state system. A vast number of methods exist for fulfilling this task.

The above reduction does not generally apply to timed automata with dense-time semantics. Since the model is inherently infinite-state, the decidability of the reachability problem is far from obvious. The difficulty has been overcome by Alur and Dill using an elegant technique: the *region graph* abstraction [9]. The idea is to partition the infinite space of clock values (and consequently, the infinite space of configurations) into a finite set of *regions* so that two configurations that belong to the same region are equivalent in terms of their possible future behaviors. The set of regions is carefully constructed so that clock vectors are in the same region iff they satisfy the same constraints and will continue to do so despite time elapsing or some of the clocks being reset.

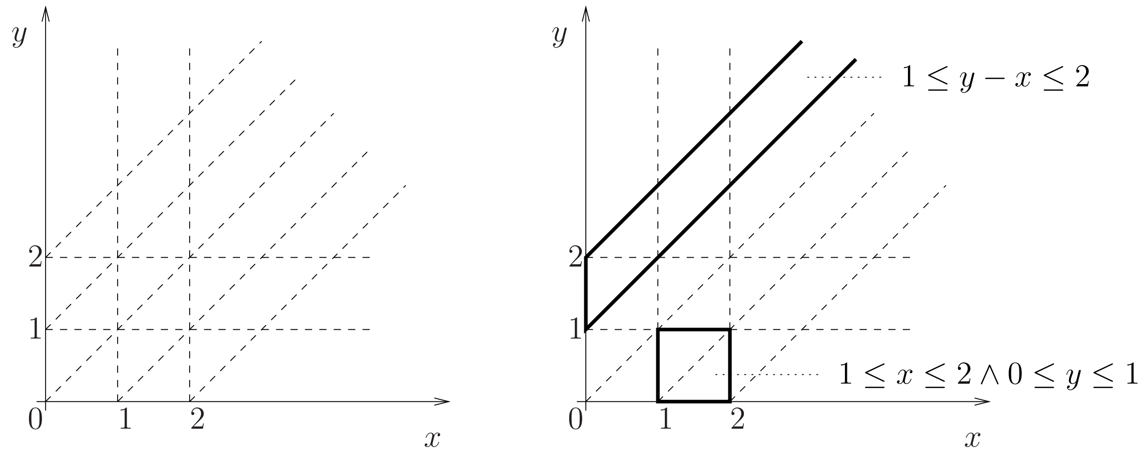

Figure 1.6: A partition of the space of two clocks x and y into 78 regions (left); two zones (right).

Regions are illustrated in Figure 1.6. The figure shows the partitioning into regions of the space of two clocks. Roughly speaking, every (x, y) point where x and y assume integer values

1. Then clocks assume integer values.

not greater than 2 is a region: e.g., the point $(x = 1, y = 1)$ is a region and so is $(x = 1, y = 0)$. Also, open straight line segments such as $x = 0 \wedge 0 < y < 1$ or $1 < x < 2 \wedge x = y$ are regions. Open triangles such as $1 < x < 2 \wedge 0 < y < 1 \wedge x < y$ are also regions. Finally, unbounded sets such as $1 < y - x < 2 \wedge x > 2$ are also regions. For an exact definition of regions, the reader is referred to [3]. To keep the number of regions bounded, the same idea as the one described above in the case of discrete-time is used, namely, abstracting the values of clocks that exceed some maximal constant c_{max}. In Figure 1.6, c_{max} is taken to be 2.

Using the concept of regions, a (dense) timed automaton can again be seen as a finite-state automaton: its states are pairs of discrete (control) states and regions. However, although the region graph is an invaluable tool for proving decidability, it is not very useful in practice. The reason is that the partition into regions is too *fine-grained*, resulting in a huge number of regions (exponential in the worst case in both the number of the clocks and the size of the constants used in the timing constraints). Keeping in mind that the size of the discrete state space (excluding the clocks) is often already very large, timed automata suffer from what can be called a “double” *state explosion* problem. Much of the research in the timed-automata community has attempted to overcome this problem by finding more efficient verification methods. Some of these attempts are described below. It is fair to say that no “silver bullet” has been found to the problem, and TA model checking is still more expensive in practice than “untimed” model checking (which is consistent with the worst-case theoretical complexity of the problem). This is still an active area of research.

One of the ideas was to find partitions that are coarser than the region graph. This led to the ideas of *time-abstract quotient* [4, 110, 106] and *zone graph* [16, 35, 105, 33, 70, 22]. The time-abstract quotient of a TA can be seen as a coarse region graph, which still has the same properties. It is obtained by “splitting” sets of configurations depending on their successors, using a classic partition-refinement method [82]. In practice, the refined sets are much coarser than regions (i.e., they are unions of many regions) although in the worst case they can be as fine as individual regions. The zone graph is based on representing sets of configurations as convex polyhedra called *zones*. A zone can be seen as a conjunction of simple linear constraints on clocks, for instance, $1 \leq x \leq 2 \wedge 0 \leq y \leq 1$. Examples of zones over two clocks x and y are shown in Figure 1.6 (zones are depicted in thick lines).

The zone graph is built by computing all successor zones of a given initial zone in a *forward* manner. Zones may "overlap", so in theory the zone graph can be exponentially larger than the region graph! In practice, however, this does not happen. In fact the zone graph is considerably smaller, and remains to this day the most efficient method of model-checking timed automata.

Both the time-abstract quotient and the zone-graph methods raise a number of interesting problems that have to do with the *symbolic* representation of sets of configurations. Dill [35] proposed an efficient method to represent zones as matrices of size $n \times n$, where n is the number of clocks. These are called *difference bound matrices* or DBMs. DBMs are used in implementations of the time-abstract quotient as well as the zone graph methods. For the former, care must be taken to ensure that partition refinement yields only convex sets of configurations, that is, zones, so that DBMs can be used [106]. In the case of the zone graph, care must be taken to ensure that the graph remains finite, and a set of abstractions have been developed for this purpose [105, 34, 22].

Reachability covers many of the properties that one usually wishes to check on a model, but not all. In particular, *liveness* properties (those that state, informally speaking, that the system indeed exhibits "good" behaviors) cannot be reduced to reachability. One way to model such properties is by means of *timed Büchi automata* (TBA). TBA are the timed version of Büchi automata: the latter define sets of infinite behaviors (rather than sets of finite behaviors, defined by standard finite automata). TBA often arise when composing a (network of) plain TA with a monitor of a liveness property: the monitor is often modeled as an "untimed" Büchi automaton, however, the composition of the TA model and the monitor yields an automaton which is both timed and Büchi. Let us provide an example.

A typical example of a liveness property is the *unbounded response property* "a is always followed by b". Notice that this is the "unbounded" version of the property modeled by the monitor of Figure 1.5. It is "unbounded" in the sense that it does not specify how much later b must occur after a. It only requires that b occurs *some* time after a has occurred. In order to check this property, we will again build a monitor that attempts to capture the violation of the property. This monitor is the (untimed) Büchi automaton shown in Figure 1.7. The monitor non-deterministically chooses to monitor an event a, moving from state 0 to state 1. If a b never occurs, the monitor has an infinite execution where it remains at accepting state 1: this is an accepting execution, meaning the property is violated. If b is received, on the other hand, the monitor moves to state 2, which is non-accepting. If no execution is accepting, the property is satisfied.

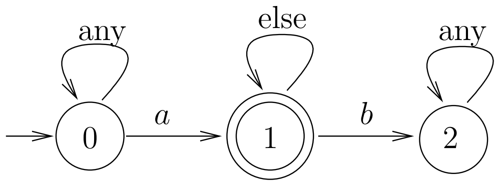

Figure 1.7: A Büchi-automaton monitor for checking an unbounded-response property.

TBA emptiness can be checked in theory on the region graph, interpreted as a finite (untimed) Büchi automaton. In practice, the time-abstract quotient graph or the zone graph can be used instead. The former can be easily shown to preserve liveness properties [106]. The fact that the zone graph can be used to check TBA emptiness is non-trivial and has been completely proven only recently [101].

1.3.2 Verification of hybrid automata

Timed automata are a very special class of hybrid automata. The decidability results for timed automata were generalized to some slightly more complex classes such as multirate automata [6, 80] and initialized rectangular automata [89]. In multirate automata, the derivatives of the continuous variables can take constant values other than 1. In rectangular automata, the derivatives are not constant and allowed to take any value inside some interval. Decidability was also proved for some particular planar systems including 2-dimensional Piecewise Constant Derivatives PCD [76], planar multipolynomial systems [26] and non-deterministic planar polygonal systems [14]. Despite these extensions, however, the reachability problem for general hybrid automata is undecidable. In fact, this holds even for classes of systems with constant derivatives, such as linear hybrid automata with 3 or more continuous variables [47].

Let us illustrate some of the issues that arise in hybrid automata reachability. We will focus only on the problem of computing the reachable sets of hybrid automata where continuous dynamics are defined by non-trivial differential equations. A major difficulty comes with the two-phase evolution of these systems, which requires the ability to compute the successors (or predecessors) of sets of states not only by discrete transitions but also by continuous dynamics. In the continuous phase, this relates to the special problem of characterizing trajectories of continuous systems. For simplicity, we consider a hybrid automaton with only one discrete state and the initial set $Init$. The initial set can be characterized by a formula $\phi_{Init}(x)$ whose truth value is 1 iff $x \in Init$. Suppose further that the differential equation $\dot{x} = f(x)$ of the continuous dynamics admits a closed-form solution $\xi_x(t)$ for every initial condition x; hence the reachable set from $Init$ is exactly the set of x for which the formula $r(x) \equiv \exists x' : \ \phi_{Init}(x') \wedge \exists t \geq 0 : \ x = \xi_{x'}(t)$ is true. Similarly, proving that the system does not reach a bad state in the set $\mathcal{B}$, represented by a formula $\phi_{\mathcal{B}}(x)$, amounts to proving that the formula

$$\forall x' : \ \phi_{Init}(x') \Rightarrow \ \forall t \geq 0 : \ \neg\phi_{\mathcal{B}}(\xi_{x'}(t)) \tag{1.1}$$

is true, which can be done by eliminating the quantifiers.

When the derivative f is constant, for example $f(x) = c$, we have $\xi_x(t) = x + ct$. For systems with constant derivatives and where invariants and guards are specified by linear inequalities (such as timed automata and linear hybrid automata) the reachable sets can be expressed by linear formulas. Therefore the quantifiers in (1.1) can be eliminated using linear algebra. A number of tools for systems with piecewise constant derivatives have been developed, such as Kronos [33], Uppaal [70], HyTech [50] and PhaVer [39]. However, the problem becomes more difficult for systems with non-trivial continuous dynamics. On one hand, in many cases we do not know explicit solutions of the differential equations. Even if we know such solutions, such as for a linear system $\dot{x} = Ax$ with a closed-form solution $\xi_x(t) = e^{At}x$, a proof of (1.1) is possible only for a very restricted class of matrices A with special eigenstructure [83, 10].

In addition, successive computations of the states reachable by the continuous dynamics and discrete transitions may not terminate, by alternating indefinitely between two or more discrete states and each time adding more and more successors. Indeed, the reachability problem is undecidable for general hybrid systems, except for the classes with the above mentioned special linear continuous dynamics and memoryless switching dynamics [83, 10].

Since, in general, there exists no exact reachable set computation method, approximate methods have been developed. In order to be able to compute the reachable sets of a hybrid automaton, we need a finite syntactic representation of these sets. The continuous state space of hybrid automata is in $\mathbb{R}^n$ and hence can only be represented *symbolically*, such as by formulas of some logic. Examples of classes of subsets of $\mathbb{R}^n$ which admit a symbolic representation are the polyhedral sets (represented by Boolean combinations of linear inequalities) and the semi-algebraic sets (represented by combinations of polynomial inequalities). Another requirement in choosing a set representation is that it can be efficiently manipulated not only for the computation of continuous successors, but also for the treatment of discrete transitions, such as set intersection for computing the states satisfying the guard conditions and the images by the resets. Polyhedra, ellipsoids and level sets are the most commonly used representations in hybrid systems reachability algorithms [45, 30, 27, 67, 20, 11, 98, 77, 40, 59, 29]. These representations have been used in a variety of tools such as Coho[45], CheckMate [27], d/dt [13], VeriShift [20], HYSDEL [99], MPT [69], HJB toolbox [77], Ellipsoidal Toolbox ET [68]. In the following, we review some techniques for reachability computation of continuous systems and their extensions to hybrid systems. The field is vast, so we can only provide a brief review and refer the reader to the current literature for a more extensive view.

In order to control the approximation error, as in numerical simulation, most reachability algorithms use a time discretization and operate on a step-by-step basis, for example $R^{k+1} = \delta(R^k, f, t_k, t_{k+1})$ where δ denotes a function that returns an approximation of all the states reachable from R^k by the dynamics $\dot{x} = f(x)$ during the time interval $[t_k, t_{k+1}]$. The quantity $h_k = t_{k+1} - t_k$ is called the *time step*. We use δ_t to denote the set of all states reachable at a *discrete time point* t and $\delta_{[t,t']}$ the set of all states reachable in dense time, that is for all time points $\tau \in [t, t']$.

Autonomous linear systems. For a system $f(x) = Ax$, if X^k is a bounded convex polyhedron represented by the convex hull of its vertices $X^k = chull\{v_1, \ldots, v_m\}$, then the set of all states reachable from X^k at exactly time t_{k+1} can be written as $X^{k+1} = \delta_{t_{k+1}} = chull\{e^{Ah_k}v_1, \ldots, e^{Ah_k}v_m\}$ where e^{At} denotes the matrix exponentiation (which is a linear operator). Then, the set $\delta_{[t_k,t_{k+1}]}$ of all states reachable from X^k *during the time interval* $[t_k, t_{k+1}]$ can be approximated by "interpolating" the sets X^k and X^{k+1}, for example $\delta_{[t_k,t_{k+1}]}(X^k) = C^{k+1} = chull(X^k \cap X^{k+1})$. In order to achieve a conservative approximation, C^{k+1} is then "bloated" by some amount ε that bounds its distance to the exact reachable set. The computation in the next step $(k+1)$ can start from X^{k+1} to compute X^{k+2}, and then from X^{k+1} and X^{k+2} to obtain the bloated convex hull C_o^{k+2} (see Figure 1.8).

This method is indeed the basis of the reachability computation technique for linear systems implemented in the tools CheckMate [27] and d/dt [13]. The tool d/dt additionally over-approximates the convex polyhedra X^k by an orthogonal polyhedron G^k, in order to accumulate all the reachable states in a single orthogonal polyhedron. Orthogonal polyhedra [21] (which can be defined as unions of closed full-dimensional hyper-rectangles) are, unlike convex polyhedra, closed under the union operation. In addition, they admit a canonical representation allowing to perform Boolean operations (in particular the union operation) more efficiently than operations on convex polyhedra. Figure 1.8 illustrates the first two iterations of this method where the initial set X^0 is a 2-dimensional segment and the first time step is r.

Linear systems with uncertain input. For a system $\dot{x}(t) = Ax(t) + u(t)$ where $u(\cdot)$ is a piecewise continuous function such that $||u(\cdot)|| \leq \mu$, and $|| \cdot ||$ is some norm on $\mathbb{R}^m$. The above described method for autonomous systems can be extended to these systems, using the *Maximum principle* from optimal control [58]. Indeed, given a state x^* on the boundary of the initial set X^0, one can determine an input function $u^*(\cdot)$ such that the trajectory from x^* under this input lies on the boundary of the reachable set.

To exploit this fact, one can represent the reachable set by its support function. The support function of a compact and convex set $X \subset \mathbb{R}^n$ is $\rho_X : \mathbb{R}^n \rightarrow \mathbb{R}$ such that for a vector $l \in \mathbb{R}^n$, $\rho(l) = max_{x \in X}\langle l, x\rangle$ (where $\langle\cdot\rangle$ denotes the inner product). Figure 1.9 illustrates this

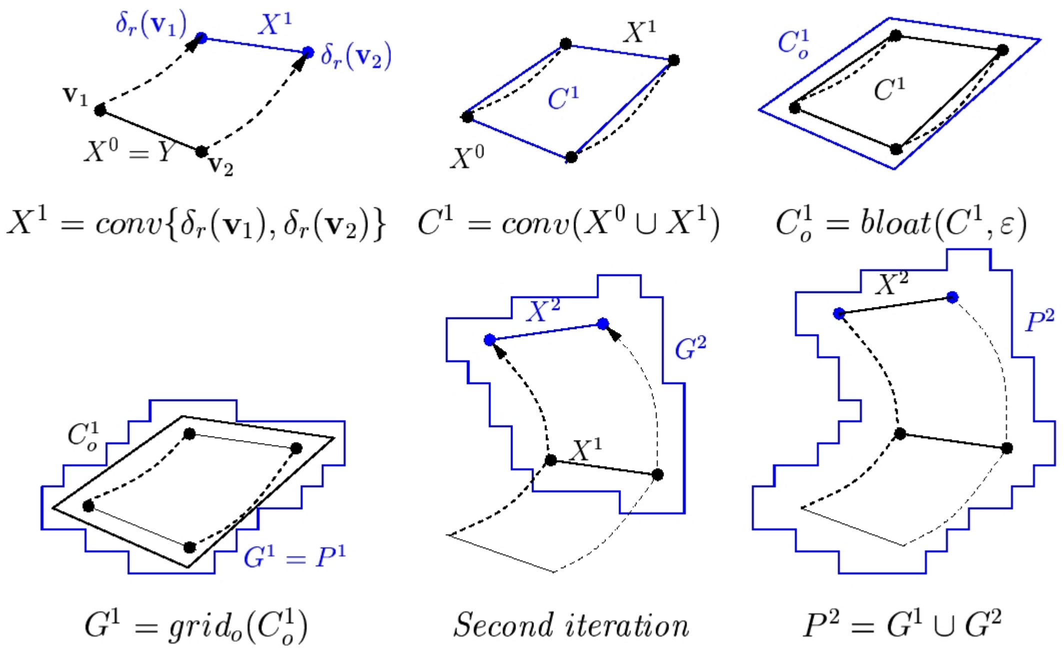

Figure 1.8: Illustration of the reachability technique for linear systems using convex polyhedra.

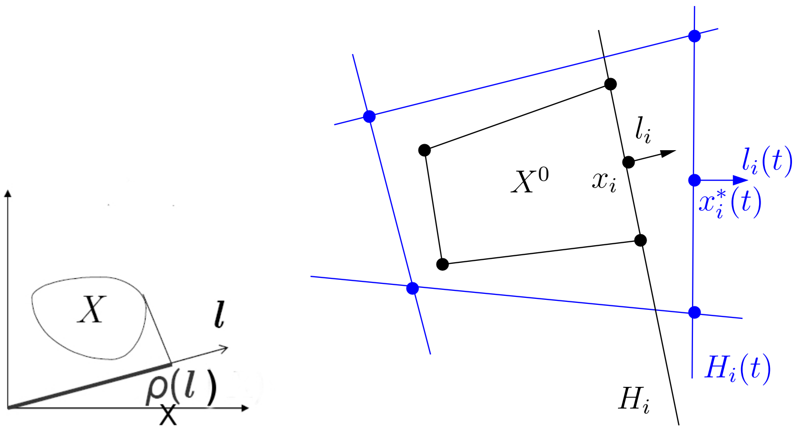

Figure 1.9: Illustration of support functions.

definition. Therefore, if X is a convex polyhedron, it can be represented as the intersection of its halfspaces $X = \cap_{i=1}^{m} H_i$ where $H_i = \{x \in \mathbb{R}^n : \langle l_i, x\rangle \leq \rho_X(l_i) = \langle l_i, x^*\rangle\}$; l_i is indeed the normal vector of the hyperplane of H_i. The point x^* lies on this hyperplane and is called the support vector of X in the direction l_i. Then, using the Maximum principle, we can find for each hyperplane H_i an input function under which the evolutions of $l_i(t)$ and $x^*(t)$ define a new hyperplane $H_i(t) = \{x \in \mathbb{R}^n : \langle l_i, x\rangle \leq \rho_X(l_i(t)) = \langle l_i(t), x^*(t)\rangle\}$. Then, from all such hyperplanes $H_i(t)$ $(i = 1, \ldots, m)$ we define a polyhedron that over-approximates the reachable set at time t.

This is also the basic principle employed by the reachability technique using ellipsoidal approximations [67, 20, 68]. An ellipsoid X can be described as $X = \{x \in \mathbb{R}^n : x^T Q^{-1} x \leq 1\}$ where Q is positive definite; its support function is $\rho_X(l) = \sqrt{l^T Q l}$ and its support vector in the direction l is $\frac{Ql}{\sqrt{l^T Q l}}$. Then, for a given point on the boundary of the ellipsoid, one can use the Maximum principle to track the time evolution of the corresponding support vector l and that of the matrix Q, in order to yield an over-approximation E_o and an under-approximation E_i of the reachable set (see Figure 1.10). This method and its extension to hybrid systems were implemented in the Ellipsoidal Toolbox ET [68].

Another way to handle the input uncertainty is to bound its effects in each time step by enlarging the reachable set of the corresponding autonomous system as follows: $X^{k+1} = e^{Ah_k} X^k \oplus B(\eta)$ where $\oplus$ denotes the Minkowski sum, and $B(\eta)$ the ball centered at the origin with the radius $\eta = \dfrac{e^{h_k}||A|| - 1}{||A||}\mu$ (μ is a bound of $||u||$). This expansion is similar to the bloating operation used in the method for autonomous systems to cover all the states reachable in dense time.

If the sets X^k are polyhedra, one can use the infinity norm and $B(\eta)$ is thus a box. The computation of the Minkowski sum may generate new sets with high geometric complexity (expressed in terms of the number of vertices and facets). Recently, [40] proposed to use zonotopes, instead of convex polyhedra, as a new set representation. A zonotope X can be described as $X = \{\sum_{i=1}^{m} \alpha_i g_i \mid \forall i \in \{1, \ldots, m\} : \alpha_i \in [-1, 1]\}$, and the vectors g_i are called *generators* (see Figure 1.11). An interesting property of zonotopes is that the Minkowski sum of two zonotopes can be obtained by taking the union of their generators. Figure 1.12

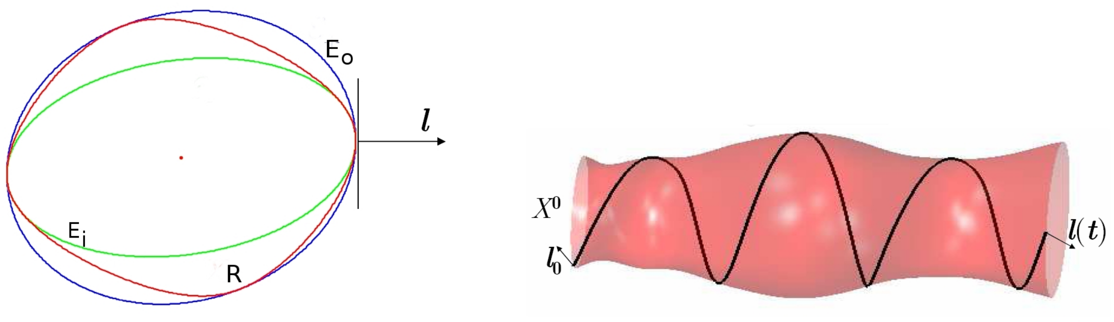

Figure 1.10: Reachability computation using ellipsoids [68].

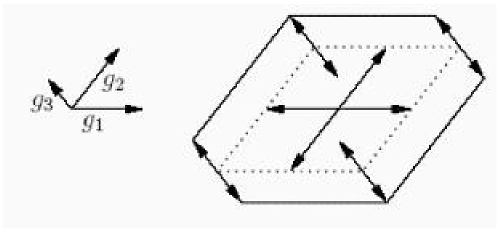

Figure 1.11: A zonotope.

illustrate one iteration of the reachability computation using zonotopes. The first zonotope (which is indeed a parallelepiped) is the initial set X^0, the second one is $X^1 = e^{Ah_0}X^0$, that is the result of applying the linear transformation e^{Ah_0} to X^0. The Minkowski sum $X^1 \oplus B$ where B is the box representing the effects of uncertainty is the last zonotope shown in the figure. One can see that the number of generators of the resulting zonotope grows iteration after iteration, but we can over-approximate it by a zonotope with a smaller number of generators [42]. The main inconvenience of this representation is that Boolean operations over zonotopes are not easy to compute. A method for computing the intersection of a zonotope and a hyperplane was proposed in [41]. Alternatively, oriented boxes can provide a good compromise between the approximation error and computational expenses [94].

Non-linear systems. While many properties of linear systems can be exploited to develop relatively efficient reachability techniques, the situation is more difficult for non-linear systems. One approach to solving this problem is to use optimization to describe the 'extremal behaviors'. In [30], in each time step, the boundary of an orthogonal polyhedron is lifted outwards by some amount that guarantees to cover all the reachable states during that

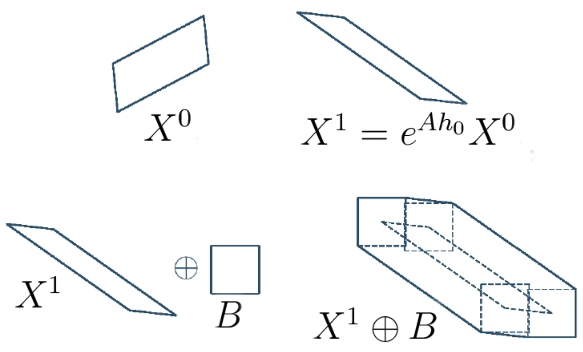

Figure 1.12: Reachability computation using zonotopes.

step, as shown in Figure 1.13. For example, a face e is lifted outwards by the amount $f_m h$ if $f_m > 0$ where f_m is the maximum of the projection of the derivative on the normal of e within some neighborhood of e, and h is the time step. The reachability technique in [27] first computes the convex hull C of the successors at time t from the vertices of the initial polyhedron X^0 (see Figure 1.13), as in the above described method for linear systems. However, unlike for linear systems, this convex hull C clearly does not include all the reachable states at t (as illustrated by the dashed trajectory in the figure). Nevertheless, the directions of its faces can be used to form a 'tight'[7] polyhedral over-approximation X^1 by estimating the distance to the exact set in the directions of the faces. A similar idea has recently been used in [91] where the reachable sets are approximated by template polyhedra (with fixed constraint matrices) using a Taylor expansion of the trajectories.

Another approach is based on a formulation of the evolution of the reachable set, represented by its level set, according a Hamilton-Jacobi partial differential equation (see for example [98]). This technique was implemented in the HJB toolbox [77].

Polynomial systems have recently received a special interest, partly because of their applications in the modeling and analysis of biological systems. A method using Bézier techniques for these systems was proposed in [29]. It exploits the fact that by choosing an appropriate basis change, one can exploit the coeficients of the representation of a polymial in order to compute the image of a set by the polynomial.

[7]if the optimization problem can be exactly solved.

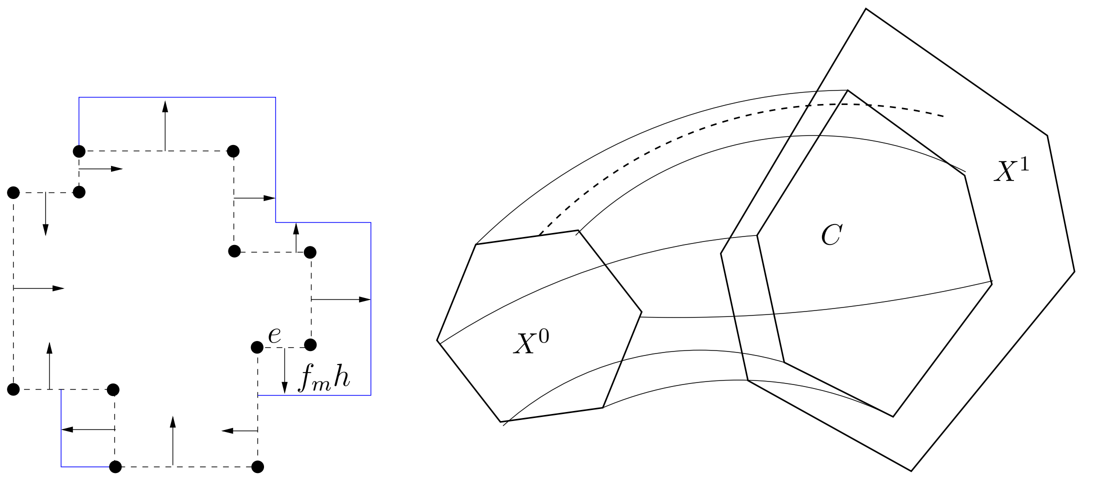

Figure 1.13: Illustration of reachability computations for non-linear systems using optimization on orthogonal polyhedra (left) and convex polyhedra (right).

Most of the above described methods are equipped with an error control mechanism, that is, they can produce an approximation as accurate as desired. Nevertheless, it is not always necessary to obtain a very accurate approximation of the reachable set (which is computationally expensive) but only sufficiently accurate to prove the property of interest. Barrier certificates [86] and polynomial invariants [97] can be seen as such approximations. A barrier certificate can be intuitively seen as a proof of the existence of an 'impermeable' frontier between the reachable set and the bad set. The method in [86] searches for such a frontier in the form of a polynomial, and the 'impermeability' can be expressed using the derivatives along the frontier. This results in an optimization problem that can be solved using the sum of squares optimization tool SOSTOOLS [87].

Abstraction. The main idea of this approach is to start with a rough (conservative and often discrete) approximation of a hybrid system and then iteratively refine it. This refinement is often local in the sense that it uses the previous analysis results to determine where the approximation error is too large to prove the property (see for example [96, 8, 28]). A popular abstraction approach is predicate abstraction where a conservative abstraction can be constructed by mapping the infinite set of states of the hybrid system to a finite set of abstract states using a set of predicates. The property is then verified in the abstract system. If it holds in the abstract system, it also holds in the concrete hybrid system. Otherwise, a counter-example can be generated. If the abstract counter-example corresponds to a concrete trajectory, then the hybrid system does not satisfy the property; otherwise, the abstract counter-example is spurious because the abstraction is too conservative, and the abstraction can then be refined to achieve a better precision.

In the following, we illustrate this approach by explaining the method using polynomials proposed in [96]. The continuous state space $\mathbb{R}^n$ is partitioned using the signs of a set of polynomials. As an example, an abstract state s defined by $g_1(x) < 0 \ \wedge \ g_2(x) > 0$ corresponds to a (possibly infinite) set $c(s)$ of concrete states. Then, the abstract transition over-approximates the concrete one such that there is a transition from s to s' if there exists a trajectory from a concrete state in $c(s)$ to another concrete state in $c(s')$. More precisely, in this method, first the set of polynomials is saturated by adding all the high-order derivatives of the initial polynomials. Then, by looking at the sign of the polynomials, it is possible to decide whether a trajectory can go from one abstract state to another. For example, if there are only two polynomials g_1 and g_2 such that $g_2 = \dot{g}_1$. Suppose that the abstract state s satisfies $g_1 = 0$ and $g_2 > 0$, then the new sign of g_1 is positive and from s we add a transition to s' satisfying $g_i > 0$. The abstraction can be refined by adding more polynomials.

Another abstraction method in [8] uses linear predicates to partition the continuous state space, and thus each abstract $c(s)$ is a convex polyhedron. The abstract transition from s to s' is determined by computing the reachable set from $c(s)$ and check whether it reaches $c(s')$. This is less expensive than the reachability computation on the hybrid system which requires handling accumulated reachable sets with geometric complexity that grows after successive continuous and discrete evolutions.

Box decompositions are also commonly used to define abstract systems, such as in [90, 59]. The abstract system can then be built by exploiting the properties of the system's vector fields over such decompositions. The method proposed in [59] makes use of the following special property of multi-affine systems:[8] the value of a multi-affine function $f(x)$ with x inside some box can be expressed as a linear combination of the values of f at the vertices of the box. Using this, one can determine whether the derivative vector on the boundary

[8]Multi-affine systems are a particular class of polynomial systems such that if all the variables x_i are constant, the derivatives are linear in x_j with j not equal to i.

of a box points outwards or inwards, in order to over-approximate the reachability between adjacent boxes.

While discrete abstractions allow benefiting from the well-developed verification algorithms for discrete systems, they might be too coarse to preserve interesting properties. Timed abstractions can be built by adding bounds on the time for the system to reach from one abstract state to another. A generalization of this idea is called *hybridization* [12] involving approximating a complex system with a simpler system, for which more efficient analysis tools are available. To this end, using a partition of the state space, one can approximate locally the system's dynamics in each region by a simpler dynamics. Globally, the dynamics changes when moving from one region to another, and the resulting approximate system behaves like a hybrid system and this approximation process is therefore called hybridization. Then, the resulting system is used to yield approximate analysis results for the original system. The usefulness of this approach (in terms of accuracy and computational tractability) depends on the choice of the approximate system. For example, the hybridization methods using piecewise affine approximate systems, proposed in [12], allows approximating a non-linear system with a good convergence rate and, additionally, preserving the attractors of the original system. In addition, the resulting approximate systems can be handled by the existing tools for piecewise affine systems (presented earlier in this section).

1.4 Partial verification

Exhaustive verification is desirable since, if it succeeds, it guarantees that a model satisfies a property. But exhaustive verification has its limitations as we have seen: state-explosion or even undecidability. In fact, state-explosion is a phenomenon that is also prevalent in the exhaustive verification of much simpler, finite-state models. This phenomenon has so far hindered a wider adoption of exhaustive verification in industrial applications, because the size of the problems tackled there is far too big to treat exhaustively. Instead, practitioners use simulation as their main verification tool.[9] Even though simulation cannot prove that a

[9]The term "verification" usually denotes simulation-based verification in industrial jargon, whereas "formal verification" is used to denote exhaustive verification.

property is satisfied, it can certainly reveal cases where it is not satisfied, that is, potential bugs of the real system, its model, or its specification.

An advantage of simulation is that it has some time-scalability properties: running 200 simulations is better (i.e., likely to discover more bugs) than running 100 simulations, and running longer simulations is also better. Moreover, if 100 simulations can be run in one day, say, then in two days we can most likely run 200 simulations. In contrast, most exhaustive verification tools suffer from a “hitting the wall” type of problem. Once they exhaust the main memory of the computer that they run on, they start using disk space, which involves a lot of swapping on the OS side. Disk swapping virtually takes all processing time, leading verification to a halt. This means that the number of new states that are explored per unit of time radically decreases to practically zero, as illustrated in Figure 1.14. Usually this wall is hit after relatively little time, in the order of minutes.[10] Then, running the tool for many hours will not improve the number of states that are explored compared to running it for ten minutes. This is not time-scalable.

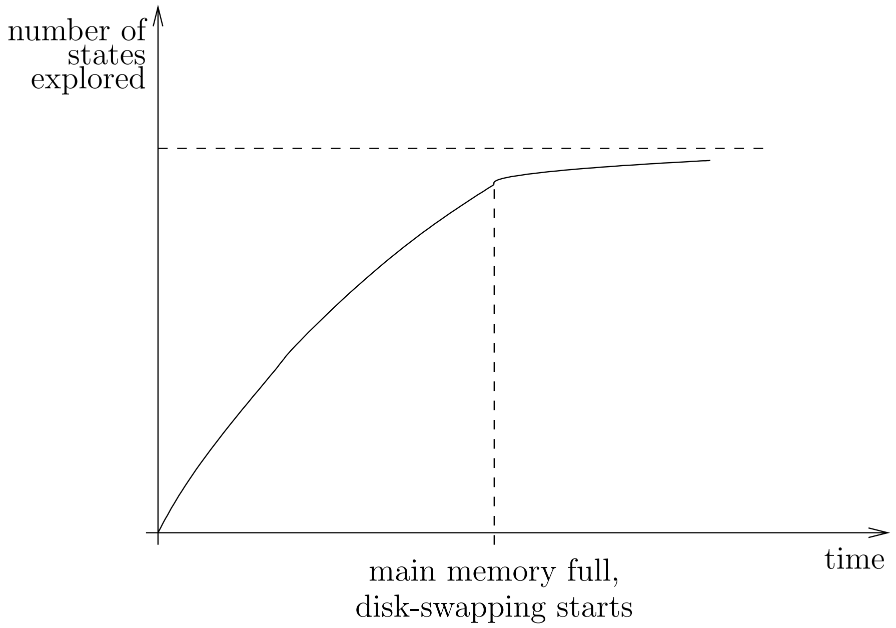

Figure 1.14: Hitting the exhaustive verification wall.

[10]For example, using a model-checker that can explore 10^5 new states per second, on a model that requires 1000 bytes to represent each state, consumes memory at a rate of approximately 100 MB/sec. This means that a main memory of size 8 GB can be filled in about 2 minutes. Exploration rates in the order of 10^5 states per second are not unusual for an advanced model-checker such as Spin [52].

Time-scalability is obviously critical in an industrial setting, where predictability in terms of allocation of resources vs. expected benefits is highly desirable. Although simulation is more time-scalable, it is still not fully predictable. Running 200 instead of 100 simulations obviously does not guarantee that twice as many bugs will be found. It does not guarantee either that twice as many states will be explored. These are some of the reasons that prompted more systematic methodologies for simulation-based verification (e.g., see [108] for the case of the hardware industry and [44] for work done in a software context). In the hardware case, these methodologies include specialized languages for writing *testbenches* (i.e., simulation environments that allow to specify input-generation policies as well as property monitors), for example, see [111, 53].

In this context, the principle of *randomization* is often used as a good aid to uncover corner cases and eventually bugs (e.g., see [75, 64, 92, 78]). We discuss some applications of the randomized state-space exploration principle to embedded system models in the rest of this section. We also introduce the concept of *resource-aware* verification, which goes beyond randomization, and includes all verification methods that are explicitly aware of their memory and time resources. Finally, we examine a particular randomized search algorithm, RRT, and its application to hybrid automata.

1.4.1 Randomized exploration and resource-aware verification

A simple technique to randomly explore a state space is *random walk*: pick randomly an initial state s_0, then pick randomly one of the successors of s_0, say s_1, then pick randomly one of the successors of s_1, and so on. This is obviously a very inexpensive algorithm in terms of space, since it needs only to store a single state at a given time, plus perhaps its set of successor states.[11]

Basic random walk is limited, especially when bugs lie very "deep" in the state space, that is, the paths to reach an error state are very long. Then, unless the number of such paths is very large, the probability to follow a path that leads to an erroneous state is

[11]If there is a number_of_successors() function available, then we don't even need to keep the set of successor states. We can just compute the number of successors, say n, then randomly choose an integer i in the interval $[1..n]$, and then replace the current state with its i-th successor.

very small. To alleviate this problem different variants of "pure" random walk have been proposed. One such variant is the *deep random search* (DRS) algorithm proposed in [46] and applied in the context of timed automata. DRS stores during the random walk a subset of the nodes it visits, called a *fringe*, and then randomly backtracks to a node in the fringe when a deadlock (a node with no successors) is reached. DRS can be applied to any model for which forward reachability is available. In the case of timed automata, the "nodes" that DRS visits correspond to symbolic states consisting of discrete state vectors plus symbolic representations of sets of clock values, using data structures such as DBMs, as explained above.

As described above, DRS maintains a fringe, which is a set of states. For "deep" random walks, this fringe can grow quite large, which means even DRS can suffer from state explosion and disk-swapping problems, like exhaustive verification methods. In order to alleviate these problems, an idea is to embed the "hard" memory constraints directly into the algorithm itself. This led to the concept of *resource-aware*, and in particular *memory-aware*, state-space exploration [104]. Memory-aware algorithms are meant to deal with the disk-swapping problem in a rather radical way: by simply using no disk memory, only main memory.

Memory-aware algorithms are by definition *memory-bounded*: they use no more than a specified amount of memory. Note, however, that not all memory-bounded algorithms are memory-aware. An example is random-walk: it is memory-bounded, since it stores a single state in memory at any given time. But it is not memory-aware, since its behavior does not generally depend on the amount of memory available. Thus, even though main memory could hold more than just one states, the random walk method does not make use of the extra space available.

Many existing verification techniques are memory-aware, including deterministic methods such as *bit-state hashing* [51], as well as randomized ones such as depth-first traversal with replacement [54]. See [104] for a detailed discussion. The idea of memory-aware verification is also exploited in [1], where a class of randomized exploration algorithms are introduced that use a parameter N representing the number of states that the algorithm is allowed to maintain at any given time during its execution. Given a model, N can be computed as follows. If R is the total size of (available) RAM memory (say in bytes) and storing a state of this model costs K bytes, then $N = \frac{R}{K}$.

Having N as an upper bound, many different randomized exploration algorithms can be tried out, depending on how two main policies are defined: how, given the current state of the algorithm, to pick which node to explore next (the *select* function), and how, given a selected node, to pick a successor of this node and update the state (the *update* function). Notice that updating the state does not necessarily mean just adding the state to the current set of visited states. Indeed, if the current set of states already holds N states, then in order to add a new state, at least one of the current states needs to be removed. There are obviously many different policies for choosing the select and update functions, and notice that randomization can be used in both functions.

It would be nice to be able to compare randomized algorithms such as the ones described above, so as to pick the "best" one for a given application. What criteria and methods can be used for carrying out such a comparison? In terms of criteria, they can be roughly classified in two classes: *performance* criteria and *coverage* criteria. The former represent the algorithm's performance (both in terms of memory and time) while the latter the algorithm's ability to "cover" the state space. We briefly discuss some criteria of this kind below.

One criterion which is perhaps a hybrid of performance and coverage is the *mean cover time*, or average time that it takes for the algorithm to visit all, or a given percentage, of the reachable states. Clearly, the smaller the mean cover time that an algorithm has (for a given percentage), the better it performs. This also means that given more time, the same algorithm is likely to cover more nodes than another algorithm with larger mean cover time. Conversely, one may also be interested in the *mean number of covered states* in a given, fixed, amount of time.

A set of criteria can be defined based on *reachability probabilities* of states. The reachability probability of a given state s can be defined as the probability that a given run of the algorithm (and its associated parameters) visits state s. Then, we could define as comparison criterion, the *minimum* reachability probability over all reachable states.

Note that the above criteria depend not only on the algorithm, but also on the structure of the state space to be explored. This state space is essentially a directed graph. Characteristics of the graph such as its diameter, the degree of its nodes, whether it is a tree, a DAG (directed acyclic graph), or a graph with cycles, and so on, will generally influence the behavior of an algorithm greatly. Because of this dependence, obtaining analytical formulas for the above criteria is a very difficult tasks. Even for simple graphs such as regular trees, it can be non-trivial [1]. On the other hand, experimental results can often be obtained much more easily, e.g., see [84, 1]. This is an exciting field of research and we expect it to become more popular in the near future, because of its high relevance in industrial practice.

1.4.2 RRTs for hybrid automata

Finding a trajectory of a hybrid automaton violating a safety property can be seen as a path planning problem in robotics, where the goal is to find feasible trajectories in some environment that take a robot from an initial point to a goal point [72]. In the following, we describe a partial verification algorithm based on RRT (Rapidly-exploring Random Trees) [71], a probabilistic path and motion planning technique with a good space-covering property. The RRT algorithm has been used to solve a variety of reachability-related problems such as hybrid systems planning, control, and verification (see for example [37, 17, 57, 25, 85] and references therein). This approach indeed can be thought of as a simulation-based verification approach. Along this line, one can mention the work on systematic simulation [56] and its extension with sensitivity analysis [36]. For some classes of stable systems, it is possible to use a finite number of simulations and a bisimulation metric to prove a safety property of a hybrid system [43].

The first part of the section will be devoted to the basic RRT algorithm. In the second part we extend the RRT algorithm to treat hybrid systems. By "the basic RRT algorithm", we mean the algorithm for a continuous system and without problem-specific optimization. For a thorough description of RRTs and their applications in various domains, the reader is referred to a survey [71] and numerous articles in the RRT literature.

```
Procedure RRT_Tree_Generation($x_{init}$, $k_{max}$)
  $\mathcal{T}.init(x_{init})$; $k = 1$
  Repeat
    $x_{goal} = \textsc{Random\_State}(\mathcal{X})$;
    $x_{near} = \textsc{Neighbor}(\mathcal{T}, x_{goal})$;
    $(u, x_{new}) = \textsc{New\_State}(x_{near}, x_{goal}, h)$;
    $\mathcal{T}.\textsc{Add\_Vertex}(x_{new})$;
    $\mathcal{T}.\textsc{Add\_Edge}(x_{near}, x_{new}, u)$;
  Until($k \geq k_{max} \ \vee \ \mathcal{B} \cap Vertices(\mathcal{T}^k) \neq \emptyset$)
```

Figure 1.15: The basic RRT algorithm.

Essentially, the RRT algorithm constructs a tree $\mathcal{T}$, the root of the which corresponds to the initial state x_{init}. Each directed edge of the tree $\mathcal{T}$ is labeled with an input selected from a set of admissible input functions. Hence, an edge labeled with u that connects the vertex x to the vertex x' means that the state x' is reached from x by applying the input u over a duration of h time, called a *time step*. When a variable time step is used, each edge is also labeled with the corresponding value of h.

In each iteration the function Random_State samples a *goal state* x_{goal} from the state space $\mathcal{X}$. We call it a goal state because it indicates the direction towards which the tree is expected to evolve. Then, a *neighbor state* x_{near} is determined as a state in the tree closest to x_{goal}, according to some pre-defined metric. This neighbor state is used as the starting state for the next expansion of the tree. The function New_State creates a trajectory from x_{near} towards x_{goal} by applying an admissible input function u for some time h. Finally, a new vertex corresponding to x_{new} is added in the tree $\mathcal{T}$ with an edge from x_{near} to x_{new}. In the next iteration, the algorithm samples a new goal state again.

Figure 1.16 illustrates one iteration of the algorithm. One can see that x_{near} is the state closest to the current goal state x_{goal}. From x_{near} the system evolves towards x_{goal} under the input u, which results in a new state x_{new} after h time.

The algorithm terminates after k_{max} iterations or until a bad state in $\mathcal{B}$ is reached. Different implementations of the functions in the basic algorithm and different choices of the metric and of the successor functions in the problem formulation result in different versions of the RRT algorithm. Note that in most versions of the RRT algorithms, the sampling distribution of x_{goal} is *uniform* over $\mathcal{X}$, and the metric ρ is the *Euclidian distance.*

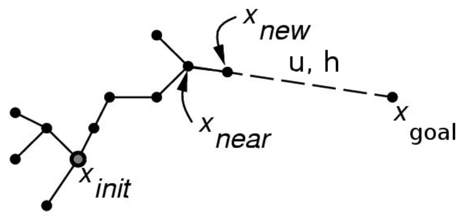

Figure 1.16: Illustration of one iteration of the RRT algorithm.

Probabilistic completeness is an important property of the RRT algorithm [65, 71], which is stated as follows: *If a feasible trajectory from the initial state x_{init} to the goal state x_{goal} exists, then the probability that the RRT algorithm finds it tends to* 1 *as the number k of iterations tends to infinity.* Although the interest of this theorem is mainly theoretical, since it is impossible in practice to perform an infinite number of iterations, this result is a way to explain the good space-covering property of the RRT algorithm.

We now describe an extension of the RRT algorithm to hybrid systems, which we call hRRT. The extension consists of the following points. Since the state space is now hybrid, sampling a state requires not only sampling a continuous state but also a discrete state.

In addition, to determine a nearest neighbor of a state, we need to define a distance between two hybrid states. In a continuous setting where the state space is a subset of $\mathbb{R}^n$, many distance metrics exist and can be used in the RRT algorithms. Nevertheless, in a hybrid setting defining a meaningful hybrid distance is a difficult problem. Finally, the successor function for a hybrid system should compute not only successors by continuous evolution but also successors by discrete evolution. In the following we briefly describe a hybrid distance, which is not a metric but proved to be appropriate for the purposes of developing guiding strategies discussed in Section 1.5.

Hybrid distance. Given two hybrid states $s = (q, x)$ and $s' = (q', x')$, if they have the same discrete component, that is, $q = q'$, we can use some usual metric in $\mathbb{R}^n$, such as the Euclidian metric. When $q \neq q'$, it is natural to use the average length of the trajectories from one to another, which is explained using an example shown in Figure 1.17. We consider a discrete path γ which is a sequence of two transitions $e_1 e_2$ where $e_1 = (q, q_1)$ and $e_2 = (q_1, q')$.

- The average length of the path γ is some distance between the image of the first guard $\mathcal{G}_{(q,q_1)}$ by the first reset function $\mathcal{R}_{(q,q_1)}$ and the second guard $\mathcal{G}_{(q_1,q')}$. The distance between two sets can be defined as the Euclidian distance between their geometric centroids. This distance is shown in the middle figure.
- The average length of trajectories from $s = (q, x)$ to $s = (q', x')$ following the path γ is the sum of three distances (shown in Figure 1.17 from left to right): the distance between x and the first guard $\mathcal{G}_{(q,q_1)}$, the average length d of the path, and the distance between $\mathcal{R}_{(q_1,q')}(\mathcal{G}_{(q_1,q')})$ and x'.

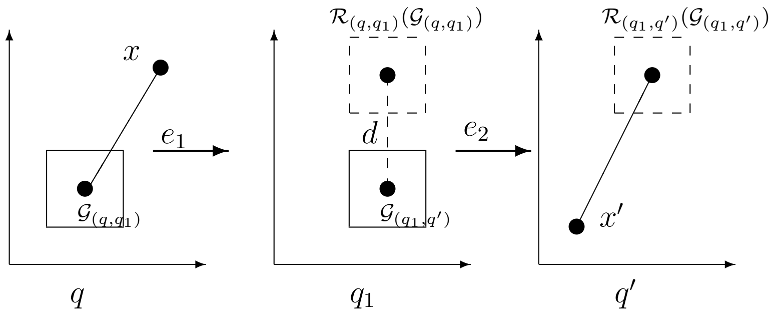
x
$\mathcal{G}_{(q,q_1)}$
e_1
q
$\mathcal{R}_{(q,q_1)}(\mathcal{G}_{(q,q_1)})$
d
$\mathcal{G}_{(q_1,q')}$
e_2
q_1
$\mathcal{R}_{(q_1,q')}(\mathcal{G}_{(q_1,q')})$
x'
q'

Figure 1.17: Illustration of the hybrid distance.

If the set $\Gamma(q, q')$ of all the discrete paths from q to q' is empty, the distance $d_H(s, s')$ from s to s' is equal to infinity. Otherwise, $d_H(s, s') = \min_{\gamma \in \Gamma(q,q')} len_\gamma(s, s')$. It is easy to see that the hybrid distance d_H is only a pseudo metric since it does not satisfy the symmetry requirement. Indeed, the underlying discrete structure of a hybrid automaton is a directed graph.

1.5 Testing

Partial verification can be termed "model testing". It is "testing" in the sense that it is generally incomplete. In this section we look at another testing activity, however, not of models, but of physical systems. In particular, we consider the following scenario: we are given a *specification* and a *system under test* (SUT) and we want to check whether the SUT satisfies the specification.

The SUT can be a software system, a hardware system, or a mix of both. Often the SUT is a *black-box* in the sense that we have no knowledge of its "internals" (i.e., how it is built). For example, if the SUT is a SW system, we have no access to the source code. If it is HW, we have no access to the HDL or other model that was used to build the circuit.[12] Instead, we can *interact* with the SUT by means of inputs and outputs: we can provide the inputs and observe the outputs. A precise, executable description of which inputs to provide and when and how to proceed depending on the observed outputs is called a *test case*. The test case is executed on the SUT[13] and at the end of the execution it outputs a PASS or FAIL verdict[14]. In the first case the SUT has passed the test, meaning that the test did not discover non-conformance to the specification (however, this clearly does not imply that the SUT meets the specification, as another test may fail). In the case of a FAIL, we know that the SUT does not meet its specification (unless the test case is itself erroneous).

In this context, the problem we are interested in is that of *test generation*, namely, synthesizing test cases automatically from a formal description of the specification. The benefits are obvious: test cases do not have to be "manually" written, which is source of errors, as with any other design process. The drawbacks of automatic test generation are similar to many other automatic synthesis techniques: state explosion problems and competition with human designers who can often "do better". In the case of testing, "better" may mean writing a *minimal* set of test cases that can cover all "important" aspects of the specification. "Minimal" can be made a formal notion (e.g., smallest number of test cases, small size of test cases, etc.). The notion of "importance" is much harder to formalize, however. Coverage has been formalized in many different ways since the beginnings of testing, in terms of statement coverage, condition coverage, and so on (e.g., see [112]). We will briefly return to some of

[12]Even if the SUT is not entirely black-box, we may still want to treat it as a black-box, because we simply have no effective way of taking advantage of the knowledge we have about its internals. For example, even if we have access to the source code of some piece of SW we want to check, we may still treat this system as black-box because we have no means of analyzing the source code (e.g., with some verification or static analysis method).

[13]Often the mere activity of executing a test case is a significant problem by itself. This is the case, for instance, when ensuring right timing on the inputs and observing accurate times of the outputs is crucial. This problem is beyond the scope of this chapter, although it is recognized as an important and practical problem.

[14]generally further verdict types may also appear, e.g., "inconclusive", "error", "none" (see the standards: UML Testing Profile by the OMG or Testing and Test Control Notation by ETSI)

these notions below.

In the next two sections, we discuss testing and test generation methods for timed and hybrid automata, respectively.

1.6 Test generation for timed automata

Before we discuss how test cases can be generated automatically from a given formal specification, we must first define what it means for an SUT to *conform* to a specification. The answer to this question depends on the setting, and over the years many different notions of conformance have been proposed by researchers. In this section, we will use a setting based on timed automata. In particular, we will use a model of timed automata with inputs and outputs (TAIO) to formally capture the specification. A TAIO is simply a TA where each one of the events labeling its transitions is distinguished to be either an input or an output (but not both). Some examples are shown in Figure 1.18. Input events are annotated with '?' and outputs with '!'. Let us look in particular at TAIO I_1. I_1 models a system that initially awaits input event a. When (and if) a is received, the system "replies" by producing output event b. The output b is produced exactly 5 time units after a was received. I_2 is similar to I_1, except that its output time is *non-deterministic*, although it is guaranteed to be no earlier than 4 and no later than 5 time units from the time a was received. (In these examples we assume that outputs implicitly have an associated notion of urgency: they can be delayed but must be eventually emitted according to the guards specified in the TAIO.) I_3 is a variant of I_2. I_4 receives a but does not "respond".

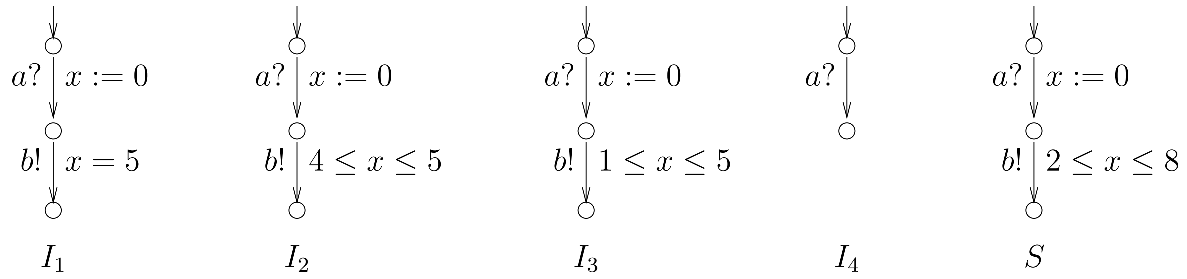

Figure 1.18: Timed automata with inputs and outputs.

To capture conformance in a formal way, we use the *timed input-output conformance* relation, or tioco, introduced in [61].[15] We illustrate tioco in the sequel through an informal description and by providing examples. A formal study can be found in [61, 60].

In Figure 1.18, TAIO I_i are given as examples of possible SUTs (they model the behaviors of such SUTs). On the other hand, TAIO S is the formal specification. S states that when/if a is received, b must be produced within 2 to 8 time units. Which of the four SUTs conform to this specification? It should be clear that I_1 and I_2 conform to S, since all their behaviors satisfy the above requirement. What about I_3? Some of its behaviors conform to S and others (the ones where b is produced earlier than 2 time units after a) do not. We therefore decide that I_3 does not conform to S: this is because it *may* produce an output too early. It should also be clear that I_4 should not conform to S: it produces no output at all.

tioco captures the above informal reasoning in a formal way. It captures the fact that the SUT is allowed to be "more output-deterministic" than the specification. Indeed, the specification generally gives some freedom in terms of what are the *legal* outputs and what are legal times that these outputs may be produced. A given SUT, which can be seen as one of the many possible *implementations* of the specification, may choose to produce any legal output, at any legal time. Different implementations will make different choices, depending on various performance, cost and other trade-offs.

An input-output system like the SUT is an *open* system, supposed to function in a given environment that generates inputs for the SUT and consumes its outputs. It is often the case that the environment is *constrained* in the sense that it does not behave in an arbitrary way. The SUT is supposed to function correctly in that sort of environment, but not necessarily in another environment, that behaves differently. For instance, a device driver for a given peripheral is supposed to work correctly only for a certain set of devices and may not work for others. An input-output specification must therefore be capable of expressing *assumptions* about the environment. Our modeling framework (and tioco) allows to capture such assumptions in an elegant way, as illustrated in Figure 1.19. The general scheme is

[15] tioco is inspired by the "untimed" conformance relation ioco introduced by Tretmans and also used in the context of testing [100]. However, tioco differs from ioco in many ways. An important difference is that tioco has no concept of *quiescence*. The latter has been included in ioco as an implicit (and somewhat problematic because it is non-quantified) way of modeling timeouts. In our setting this is unnecessary because time is a "first-class citizen" in our model.

shown to the left of the figure. It consists in modeling the specification as two separate, but communicating, TAIO models. One model captures the requirements on the SUT, that is, the guarantees that the SUT provides on its outputs: this model can be built in such a way that it is *receptive* to any input at any given time, although it may of course ignore inputs that are "illegal" or arrive at "illegal" times. The other model captures the assumptions that the SUT makes on its inputs: these specify formally which inputs are legal and at what times. These assumptions must be satisfied by the environment of the SUT.

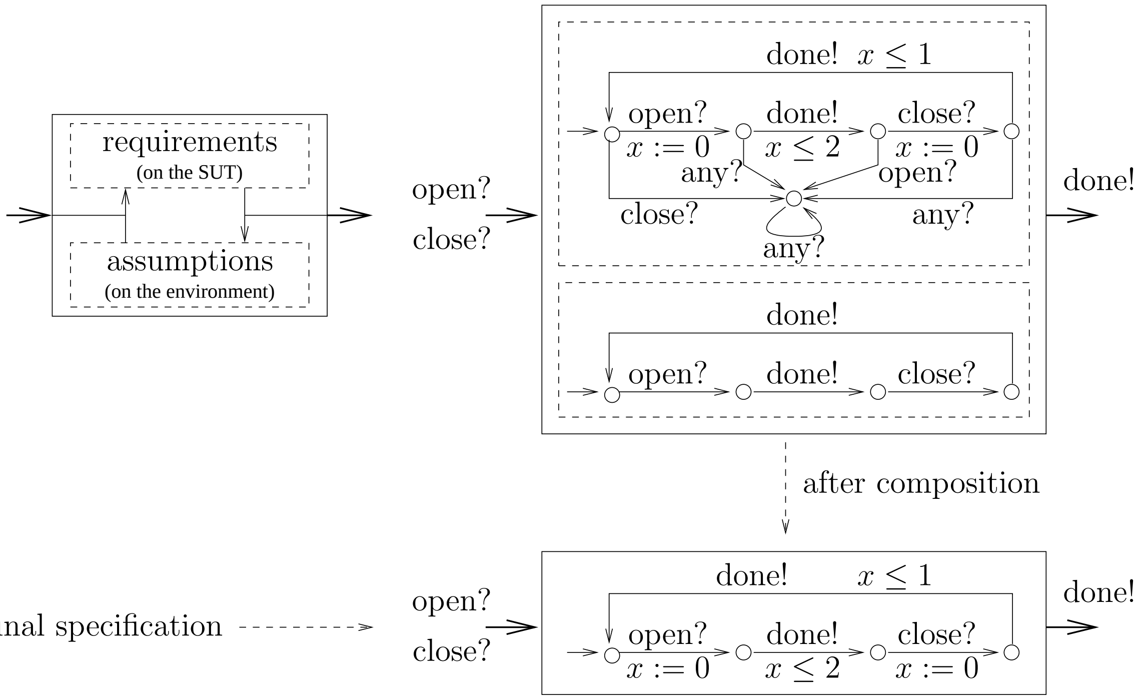

Figure 1.19: Specification including assumptions on the environment: generic scheme (left) and example (right).

We illustrate how this works more precisely through the example shown to the right of Figure 1.19. The specification concerns a system that is supposed to receive a sequence of requests to open or close, say a file. The system executes each request in a given amount of time: it takes at most 2 time units to open and at most 1 time unit to close. During that time, no new request should be received. Also, every open request should be followed by a close before a new open can be issued, and the first request should be an open.

All these assumptions on the environment are captured formally by the untimed automaton shown in Figure 1.19. This automaton is composed with the input-receptive TAIO shown at the top right, to yield the TAIO shown at the bottom right. The latter represents the final specification. Notice that this final specification is not an input-receptive TAIO: for instance, it does not accept a second open? request until the first one is fulfilled by issuing output done!. Having non-input-receptive specifications is essential in order to model assumptions on the environment, and this is an important feature of the tioco framework.

We could spend many more pages discussing what other properties are desirable from a formal conformance relation such as tioco: transitivity (if A conforms to B and B conforms to C then A conforms to C), compositionality (if A_1 conforms to B_1 and A_2 conforms to B_2, then the composition of A_1 and A_2 conforms to the composition of B_1 and B_2). These properties are satisfied by tioco, under appropriate conditions. We refer the reader to [60] for an in-depth technical study.

Having explained what it means for an SUT to conform to a formal specification, we are almost ready to discuss test generation. However, before doing that, we still need to make more precise what exactly we mean by a test case. A test case is essentially a program that is executed by a *tester*. The tester interacts with the SUT through the IO interface of the latter. The tester can be seen as a generic device, capable of running many test cases. So the tester is essentially a computer, with appropriate IO capabilities for the class of SUTs we are interested in.

The execution of a test case by the tester must be as deterministic as possible. This is crucial in order for tests to be reproducible, which in turn is very important for debugging (it is a nightmare to know that some test has failed without being able to reproduce this failure). Non-determinism can be allowed, for instance, one may allow randomized testing where some choices of the tester can be based on tossing a random coin. In reality, however, this randomness will be generated by a pseudo-random number generator, and the seed of this generator can be saved and reused to achieve reproducibility.

Obviously, determinism of the execution does not depend only on the tester (and the test case) but also on the SUT: if the SUT itself is non-deterministic (i.e., for the same sequence of inputs, it may produce different sequences of outputs) then determinism cannot be guaranteed. Still, we will require as a minimum the tester/test case to be deterministic (or random but using a pseudo-random generator as explained above).

Notice that the behavior of the SUT depends not only on the inputs it receives from the tester, but also on its internal state. In the context of this work, we will assume that it is possible to *reset* the state of the SUT to some given initial state (or set of possible states) after each test case has been executed. A large amount of research is available in the literature for the case where the SUT cannot be reset and "resetting" input sequences must be devised, in addition to the conformance testing sequences (see [73] for an excellent survey). Very little work has been done about this problem in the context of timed automata [63].

Given that a test case is a deterministic program, what does this program do? It essentially interacts with the SUT through inputs and outputs: it generates and issues the inputs to the SUT and consumes its outputs. Since the specification defines not only the legal values of inputs and outputs but also their legal timing, it is very important that the test case be able to capture timing as well. In other words, the test case must specify *not only which input should be generated but also exactly when.* Also, the test case must specify *how to proceed depending on what output the SUT produces but also on the time in which this output is produced.*

For example, consider a specification for a computer mouse that states: "if the mouse receives two consecutive clicks (the input) in less than 0.2 seconds then it should emit a double-click event to the computer". One can imagine various tests that attempt to check whether a given SUT satisfies the above specification (and indeed behaves as a proper mouse). One test may consist in issuing two consecutive clicks 0.1 seconds apart, and waiting to see what happens. If the SUT emits a double-click then it passes the test, otherwise it fails. But there are obviously other tests: issuing two clicks 0.15 seconds apart, or 0.05 seconds apart, etc. Also, one may vary the initial waiting time, before issuing the clicks. Also, presumably the specification requires that the mouse continues to exhibit the same behavior not only the first time it receives two clicks, but also every time after that. Then a test could try to issue two sets of two clicks and check that the SUT processes both of them correctly. It becomes clear that a finite number of tests cannot ensure that the SUT is correct, at least

```
// test case pseudo-code:
s := initialize state; // this is the state of the tester
while( not some termination condition  ) do
  x := select input in set of legal inputs given s;
  issue x to the SUT;
  set timer to TIMEOUT;
  wait until timer expires or SUT produces an output;
  if ( timer expired ) then
    s := update state s given TIMEOUT;
  end if;
  if ( SUT produced output y, T time units after x ) then
    s := update state s given T and y;
  end if;
  if ( s is not a legal state ) then
    announce that the SUT failed the test and exit;
  end if;
end while;
announce that the SUT passed the test and exit;
```

Figure 1.20: Generic description of a test case.

not in the absence of more assumptions about the SUT. It is also interesting to note some inherent ambiguities in the above, simple, specification written in English. For instance, does the delay between the two ticks need to be strictly less than 0.2 seconds or can it be exactly 0.2 seconds? How much time after the ticks should the mouse respond by emitting an event to the computer? And so on.

In general, a test case in our setting can be cast into the form shown in Figure 1.20. The test case is described in pseudo-code. The test case maintains an internal state, which captures the "history" of the execution (e.g., what outputs have been observed, at what times, and so on). The state can also be used to encode whether this history is legal, that is, meets the specification. If it does not, the test stops with the result FAIL. Otherwise, the test can proceed for as long as required.

The test case uses a *timer* to measure time. This timer is an abstract device that can be implemented in different ways in the execution platform of the tester. An important question, however, is what exactly can this timer measure, especially, how precise this timer measures time. For instance, in the pseudo-code, the timer is set to expire after `TIMEOUT` time units. One may ask: how critical is it that the timer expires *exactly* after so much time? What if it actually expires a bit late or a bit early? In the pseudo-code, the timer is checked to see how much time elapsed from event `x` until event `y`: this amount is `T` time units. But if the timer is implemented as an integer counter, which is typically the case in a digital computer, the value `T` that the counter reads at any given moment in time is only an approximation of the time that has elapsed since the timer was reset: in reality, the time that has elapsed lies anywhere between `T` and `T+1` time units. To the above must be added inaccuracies because of processing delays. For example, executing the tester code takes time: this time must be accounted for when updating the state of the tester.

In order to make the issues of time accuracy explicit, we make a distinction between *analog-clock* and *digital-clock* testers (and tests). The former are ideal devices (or programs), assumed to be able to measure time exactly, with an infinite degree of precision. In particular they can be assumed to measure any delay which is a non-negative rational number. Digital-clock tests have access to a digital clock with finite precision. This clock may suffer from drift, jitter, etc. Analog-clock tests are not implementable since clocks with infinite precision do not exist in practice. Still, it is worth studying analog-clock tests not only because of theoretical interest, but also because they can be used to represent ideal, or "best case" tests, that are independent from a given execution platform. This is obviously useful for test reusability. Analog-clock tests may also be used correctly when real clock inaccuracies or execution delays can be seen as negligible compared to the delays used in the test.

We are now in a position to discuss automatic test generation. The objective is to generate, from a given formal specification, provided in the form of a TAIO, one or more test cases, that can be represented as programs written in some form similar to the pseudo-code presented above. We briefly describe this quite technical step and illustrate the process through some examples. We refer the reader to [60] for a thorough presentation.

We first describe analog-clock test generation. Suppose the specification is given as a TAIO S. The basic idea is to generate a program that maintains in its memory (the `state` variable in the pseudo-code shown in Figure 1.20) the set of all possible legal configurations that S could be in, given the history of inputs and outputs (and their times) so far. Let C be this set of legal configurations. The important thing to note is that C completely captures the set of *all legal future behaviors*. Therefore, it is sufficient to determine the future of the test.

The set C is represented symbolically, in much the same way as for reachability analysis used for timed automata model-checking. C is generally non-convex and cannot be represented as a single zone, however, it can be represented as a set of zones. C is updated based on the observations received by the test: these observations are events (inputs or outputs) and time delays. Updating C amounts to performing an *on-the-fly subset construction*, which can be reduced to reachability. This technique was first proposed in [102] where it was used for monitoring in the context of fault diagnosis. The same technique can be applied to testing with very minor modifications.

Notice that the above test generation technique is *on-the-fly* (also sometimes called *on-line*). This means that the test state (i.e., C) is generated during the execution of the test, and not a-priori. There are good reasons for this in the case of timed automata: since the set of possible configurations of a TA is infinite, the set of all possible sets of legal configurations is also infinite, thus cannot be completely enumerated.

We illustrate analog-clock on-the-fly test generation and execution on the example specification S shown in Figure 1.18. Suppose the three states of S are numbered 0,1,2, from top to bottom. The initial set of legal configurations can be represented by the predicate $C_0 : s = 0$. Notice that the value of the clock x is unimportant in this case. Next, the test can choose to issue the single input event a to the SUT. The set of legal configurations then becomes $C_1 : s = 1 \wedge x = 0$. Let us suppose that `TIMEOUT`=2. If the SUT produces output b before the timer expires (i.e., in < 2 time units after it received input a), the set of legal configurations becomes empty: this is because there is no configuration in C_1 that can perform a b after < 2 time units. An empty C is an illegal state for the test: this means that the SUT fails the test in this case. Indeed, this is correct, since the SUT produces output b too early. On the other hand, if the timer expires before b is received, then C is updated to $C_2 : s = 1 \wedge x = 2$. The timer is reset, and execution continues. Suppose b is not received after four timeouts: the value of C at this point is $C_5 : s = 1 \wedge x = 8$. If a fifth timeout occurs, C becomes empty: this is because there is no state in C_5 that can let 10 time units elapse (because of the urgency implied when $x = 8$). Again, the SUT fails in this case, because it does not produce a b by the required deadline.

On-the-fly generation is not the only possibility in the case of analog-clock tests. Another option is to generate analog tests off-line, and represent them as TAIO themselves. However, these TAIO need to be deterministic, and synthesis of deterministic TA that are in some sense equivalent to a non-deterministic TA can be an undecidable problem [103]. Indeed, test generation of TA testers is generally undecidable. Still, it is possible to restrict the problem so that it becomes decidable. One way of doing this is by limiting the number of clocks that the tester automaton can have. We refer the reader to [62, 60] for details.

We now turn to digital-clock test generation. Again, we are given a formal specification in the form of a TAIO S. But in this case, we assume that we are also given a model of the digital clock, also in the form of a timed automaton. The latter is a special TA model, called a *Tick* automaton. The reason is that this TA has a single event, named *tick*, that represents the discrete tick of a digital clock (e.g., the incrementation of the digital-clock counter). Some possible Tick models are shown in Figure 1.21. The left-most automaton models a perfectly periodic clock with period 1 time unit. The automaton in the middle models a clock with drift: its period varies non-deterministically between 0.9 and 1.1 time units. In this model, the k-th tick can occur anywhere in the interval $[0.9k, 1.1k]$: as k grows, the uncertainty becomes larger and larger. The right-most automaton models a digital clock where this uncertainty is bounded: the k-th tick can occur in the interval $[k - 0.1, k + 0.1]$.

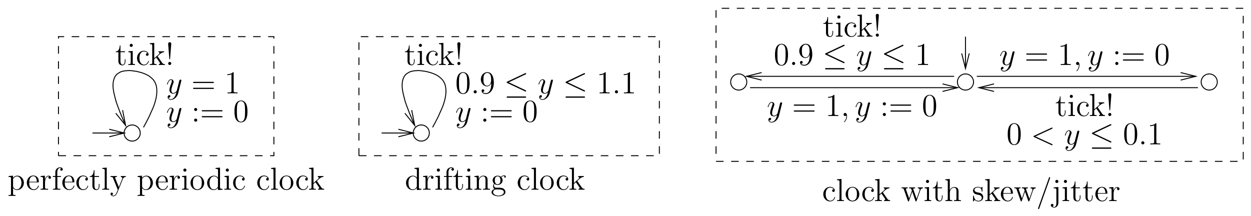

Figure 1.21: Models of digital clocks.

With a Tick model, the user has full control over the assumptions used by the digital-clock test generator. The generator need not make any implicit assumptions about the behavior of the digital clock of the tester: all these assumptions are captured in the Tick model. In an application setting, a library of available Tick models could be supplied to the user to choose a model from.

Having the specification S and the Tick automaton, automatic digital-clock test generation proceeds as follows. First, the product of S and Tick is formed, as illustrated in Figure 1.22: this product is again a TAIO, call it $S+$. The tick event is considered an output in $S+$. Next, an "untimed" test is generated from $S+$. An untimed test is one that reacts only to discrete events, and not time. However, time is implicitly captured in $S+$ through the tick event. Indeed, the tick event represents time elapse as measured with a digital clock! This "trick" allows to turn digital-clock test into an "untimed" test generation problem. The latter can be solved using standard techniques, such as those developed in [100]. Although these techniques have been originally developed for untimed specifications, they can be applied to TAIO specifications such as $S+$, because they are based on reachability analysis. Again, the idea of on-the-fly subset construction is used in this case.

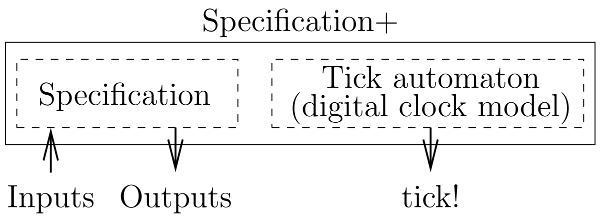

Figure 1.22: Specification+ : product of Specification and Tick model.

Let us illustrate this process through an example. Suppose we want to generate digital clock tests from the specification S shown in Figure 1.18 and the left-most, perfectly periodic, Tick model shown in Figure 1.21. A digital-clock test generated by the above method for this example is shown in Figure 1.23. Notice that the test is represented as an *untimed* automaton with inputs and outputs. This is normal, since any reference to time is replaced by a reference to the tick event of the digital clock. Also notice that inputs and outputs are reversed for the test: a is an output for the test (an input for the SUT), while b and tick are inputs to the test.

The test of Figure 1.23 starts by issuing a after some non-deterministic number of ticks (strictly speaking, this is not a valid test since it is non-deterministic: however, it can be seen as representing a set of tests rather than a single test). It then waits for a b. If a b is

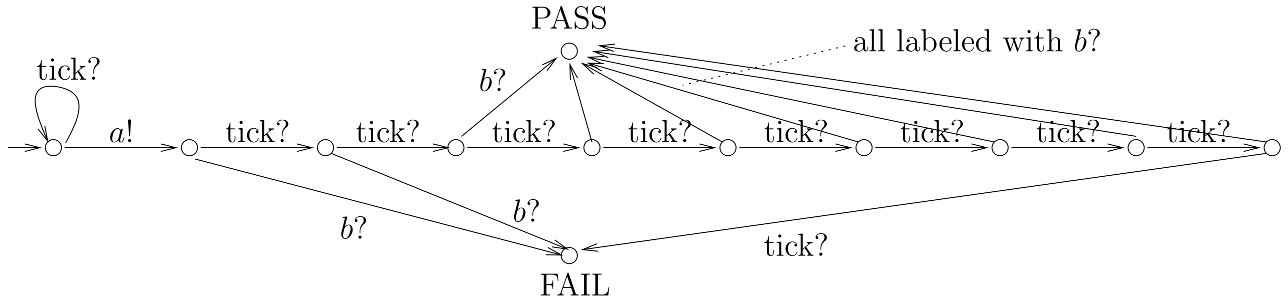

Figure 1.23: A digital clock test generated from the specification S shown in Figure 1.18 and the left-most, perfectly periodic, Tick model shown in Figure 1.21.

received before at least two ticks are received, the SUT fails the test: indeed, this is because it implies that < 2 time units have elapsed between the a and the b, which violates the specification S.[16] If no b is received for 9 ticks, the SUT fails the test: this is because it implies that > 8 time units have elapsed between a and b, which again is a violation of S. Otherwise, the SUT passes the test.

We end this section with a few remarks on test selection. The test generation techniques presented above can generate a large number of tests: indeed, in the case of multiple inputs, the test must choose which input to issue to the SUT. It must also choose when to issue this input (this can be modeled as a choice between issuing an input or waiting). This represents a huge number of choices, thus a huge number of possible tests that can be generated. This is similar to the state explosion problem encountered in model-checking: in this case it can be called the test explosion problem. It should be noted that this problem arises in any automatic test generation framework, and not just in the timed or hybrid automata case.

The question then becomes, can this large number of tests be somehow limited? Traditionally, there have been different approaches to achieve this. One approach is based on the notion of *coverage*: the idea is to generate a "representative" set of tests. One way of defining "representative" is by means of coverage: a set of tests that "covers" all aspects of the specification, in some way. In practice, notions such as state coverage (cover all states of the specification), transition coverage (cover all transitions), etc., can be used. Test generation with coverage objectives is explored in [60]. Another approach to limit test explosion

[16] We assume here that a, b, and tick cannot occur simultaneously. If this assumption is lifted, then the digital-clock test shown in Figure 1.23 needs to be modified to issue a PASS if b is received exactly one tick after a.

is to "guide" the generation of the test towards some goal: this is done using a so-called *test purpose*. A test purpose can specify, for instance, that one of the states of the specification should be reached during the test. The test generator should then produce a test that attempts to reach that state (sometimes reaching a given state cannot be guaranteed since it generally depends on the outputs that the SUT will produce). This approach has been followed by untimed test generation tools such as TGV [38] and can be easily adapted to the timed automata case.

1.7 Test generation for hybrid automata

Concerning hybrid systems, model-based testing is still a new research domain. Previous work [95] proposed a framework for generating test cases by simulating hybrid models specified using the language CHARON [7]. In this work, the test cases are generated by restricting the behaviors of an environment automaton to yield a deterministic testing automaton. A test suite can thus be defined as a finite set of executions of the environment automaton. In [57], the testing problem is formulated as one of finding a piecewise constant input that steers the system towards some set, which represents a set of bad states of the systems. The paper [55] addresses the problem of robust testing by quantifying the robustness of some properties under parameter perturbations. This work also considers the problem of how to generate test cases with a number of initial state coverage strategies.

In this section we present a formal framework for conformance testing of hybrid automata (including important notions such as conformance relation, test cases, coverage). We then describe a test generation method, which is a combination of the RRT algorithm (presented in Section 1.4.2) and a coverage-guided strategy.

Conformance relation

To define the conformance relation for hybrid automata, we need first the notions of inputs. A system input that is controllable by the tester is called a *control input*; otherwise, it is called a *disturbance input*.

Continuous inputs. All the continuous inputs of the system are assumed to be controllable. Since we want to implement the tester as a computer program, we are interested in piecewise-constant continuous input functions (a class of functions from reals to reals which can be generated by a computer). Hence, a *continuous control action* $(\bar{u}_q, h)$, where $\bar{u}_q$ is the value of the input and h is the *duration*, specifies that the system continues with the continuous dynamics at discrete state q under the input $u(t) = \bar{u}_q$ for exactly h time. We say that $(\bar{u}_q, h)$ is *admissible at* (q, x) if the input function $u(t) = \bar{u}_q$ for all $t \in [0, h]$ is admissible starting at (q, x) for h time.

Discrete inputs. The discrete transitions are partitioned into controllable and uncontrollable discrete transitions. Those that are controllable correspond to discrete control actions, and the others to discrete disturbance actions. The tester emits a discrete control action to specify whether the system should take a controllable transition (among the enabled ones) or continue with the same continuous dynamics. In the latter case, it can also control the values assigned to the continuous variables by the associated reset map. For simplicity of explanation, we will not consider non-determinism caused by the reset maps. Hence, we denote a discrete control action by the corresponding transition, such as (q, q').

We then need the notion of *admissible control action sequence*, which is formally defined in [32]. Intuitively, this means that an admissible control action sequence, when being applied to the automaton, does not cause it to be blocked.

In the definition of the conformance relation between a system under test $\mathcal{A}_s$ and a specification $\mathcal{A}$, the following assumptions about the inputs and observability are used:

- All the controllable inputs of $\mathcal{A}$ are also the controllable inputs of $\mathcal{A}_s$.
- The set of all admissible control action sequences of $\mathcal{A}$ is a subset of that of $\mathcal{A}_s$. This assumption assures that the system under test can admit all the control action sequences that are admissible by the specification.
- The discrete state and the continuous state of $\mathcal{A}$ and $\mathcal{A}_s$ are observable.

Intuitively, the system under test $\mathcal{A}_s$ is conform to the specification $\mathcal{A}$ iff under every admissible control sequence, the set of all the traces of $\mathcal{A}_s$ is included in that of $\mathcal{A}$. The definition of conformance relation can be easily extended to the case where only a subset of continuous variables are observable by projecting the traces on the observable variables. However, extending this definition to the case where some discrete states are unobservable is more difficult since this requires identifying the current discrete state in order to decide a verdict.

Test cases and test executions

In our framework, a *test case* is represented by a tree where each node is associated with an observation and each path from the root with an observation sequence. Each edge of the tree is associated with a control action. A physical *test execution* can be described as follows:

- The tester applies a test to the system under test.
- An observation (including both the continuous and discrete state) is measured at the end of *each* continuous control action and after *each* discrete (disturbance or control) action.

This procedure leads to an observation sequence, or a set of observation sequences if multiple runs of the test case are possible (in case non-determinism is present). In the following, we focus on the case where each test execution involves a single run of a test case. It is clear that the above test execution process uses a number of implicit assumptions, such as observation measurements take zero time, and in addition, no measurement error is considered. Additionally, the tester is able to realize exactly the continuous input functions (which is often not possible in practice because of actuator imprecision).

After defining the important concepts, it now remains to tackle the problem of generating test cases from a specification model. A hybrid automaton may have an infinite number of infinite traces; however, the tester can only perform a finite number of test cases in finite time. Therefore, we need to select a finite portion of the input space of $\mathcal{A}$ and test the conformance of $\mathcal{A}_s$ with respect to this portion. The selection is done using a coverage criterion that we formally define in the following. Hence, our testing problem is formulated so as to automatically generate a set of test cases from the specification hybrid automaton to satisfy this coverage criterion.

Test coverage

The test coverage we describe here is based on the star discrepancy notion and motivated by the goal of testing reachability properties. It is thus desirable that the test coverage measure can describe how well the states visited by a test suite represent the reachable set. One way to do so is to look at how well the states are equidistributed over the reachable set. However, the reachable set is unknown, we can only consider the distribution of the visited states over the state space (which can be thought of as the potential reachable space).

The star discrepancy is a notion from statistics often used to describe the "irregularities" of a distribution of points with respect to a box. Indeed, the star discrepancy measures how badly a point set estimates the volume of the box.The popularity of this measure is perhaps related to its usage in quasi-Monte Carlo techniques for multivariate integration (see for example [15]).

Let P be a set of k points inside $\mathcal{B} = [l_1, L_1] \times \ldots \times [l_n, L_n]$. Let $\mathcal{J}$ be the set of all sub-boxes J of the form $J = [l_1, \beta_1] \times \ldots \times [l_n, \beta_n]$ with $\beta_i \in [l_i, L_i]$ for all $i \in \{1, \ldots, n\}$ (see Figure 1.24). The local discrepancy of the point set P with respect to the sub-box J is defined as $D(P, J) = |\dfrac{A(P, J)}{k} - \dfrac{vol(J)}{vol(\mathcal{B})}|$ where $A(P, J)$ is the number of points of P that are inside J, and $vol(J)$ is the volume of the box J. Then, the star discrepancy of a point set P with respect to the box $\mathcal{B}$ is defined as: $D^*(P, \mathcal{B}) = sup_{J \in \mathcal{J}} D(P, J)$. The star discrepancy satisfies $0 < D^*(P, \mathcal{B}) \leq 1$. A large value $D^*(P, \mathcal{B})$ means that the points in P are not much equidistributed over $\mathcal{B}$.

Since a hybrid automaton can only evolve within the invariants of its discrete states, one needs to define a coverage with respect to these sets. For simplicity, all the staying sets are assumed to be boxes. For a set of $\mathcal{P} = \{(q, P_q) \mid q \in Q \ \wedge \ P_q \subset \mathcal{I}_q\}$ be the set of

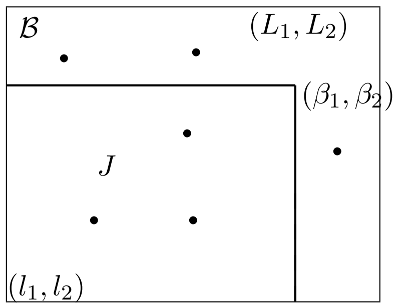

Figure 1.24: Illustration of the star discrepancy notion.

states. The coverage of $\mathcal{P}$ is defined as: $Cov(\mathcal{P}) = \frac{1}{||Q||}\sum_{q\in Q} 1 - D^*(P_q, \mathcal{I}_q)$ where $||Q||$ is the number of discrete states in Q. If an invariant set $\mathcal{I}_q$ is not a box, one can take the smallest oriented box that encloses it and apply the star discrepancy definition to that box after an appropriate coordination change. We can see that a large value of $Cov(\mathcal{P})$ indicates a good space-covering quality. The star discrepancy is difficult to compute especially for high dimensions; however it can be approximated (see [79]). Roughly speaking, the approximation considers a finite box decomposition instead of the infinite set of sub-boxes in the definition of the star discrepancy.

Coverage guided test generation

Essentially, the test generation algorithm consists of the following two steps:

- From the specification automaton $\mathcal{A}$, generate an exploration tree using the hRRT algorithm and a guiding tool, which is based on the above described coverage measure. The goal of the guiding tool is to bias the evolution of the tree towards the interesting region of the state space, so that to rapidly achieve a good coverage quality.
- Determine the verdicts for the executions in the exploration tree, and extract a set of test cases interesting with respect to the property to verify.

The motivation of the guiding method is as follows. Because of the uniform sampling of goal states, the RRT algorithm is biased by the Voronoi diagram of the vertices of the tree.

If the actual reachable set is only a small fraction of the state space, the uniform sampling over the whole state space leads to a strong bias in selection of the points on the boundary of the tree, and the interior of the reachable set can only be explored after a large number of iterations. Indeed, if the reachable was known, sampling within the reachable set would produce better coverage results.

Coverage-guided sampling. Sampling a goal state $s_{goal} = (q_{goal}, x_{goal})$ in the hybrid state space $\mathcal{S}$ consists of the following two steps: (1) Sample a goal discrete state q_{goal}, according to some probability distribution; (2) Sample a continuous goal state x_{goal} inside the invariant set $\mathcal{I}_{q_{goal}}$.

In each iteration, if a discrete state is not yet well explored, that is, its coverage is low, we give it a higher probability to be selected. Let $\mathcal{P} = \{(q, P_q) \mid q \in Q \wedge P_q \subset \mathcal{I}_q\}$ be the current set of visited states, one can sample the goal discrete state according to the following probability distribution: $Pr[q_{goal} = q] = \dfrac{1 - Cov(\mathcal{P}, q)}{||Q|| - \sum_{q' \in Q} Cov(\mathcal{P}, q')}$. Suppose that we have sampled a discrete state $q_{goal} = q$. Since all the staying sets are boxes, and the staying set $\mathcal{I}_q$ is denoted by the box $\mathcal{B}$ and called the bounding box.

As mentioned earlier, the coverage estimation is done using a box partition of the state space $\mathcal{B}$, and sampling of a continuous goal state can be done by two steps: first, sample a goal box $\boldsymbol{b}_{goal}$ from the partition, then *uniformly* sample a point x_{goal} in $\boldsymbol{b}_{goal}$. Guiding is thus done in the goal box sampling process by defining, at each iteration of the test generation algorithm, a probability distribution over the set of the boxes in the partition. Essentially, we favor the selection of a box if adding a new state in this box allows to improve the coverage of the visited states. This is captured by a potential influence function, which assigns to each elementary box $\boldsymbol{b}$ in the partition a real number that reflects the change in the coverage if a new state is added in $\boldsymbol{b}$. The current coverage estimation is given in form of a lower and an upper bound. In order to improve the coverage, both the lower and the upper bounds need to be reduced (see more detail in [32]).

The hRRT algorithm for hybrid automata in which the goal state sampling is done using this coverage-guided method is now called the gRRT algorithm (which means “guided hRRT"). To illustrate the coverage-efficiency of gRRT, Figure 1.25 shows the results obtained by hRRT and gRRT on a linear system after 50000 iterations. We can see that the gRRT algorithm has a better coverage result. Indeed with the *same number of states*, the states visisted by gRRT are more equi-distributed over the reachable set than those visisted by hRRT.

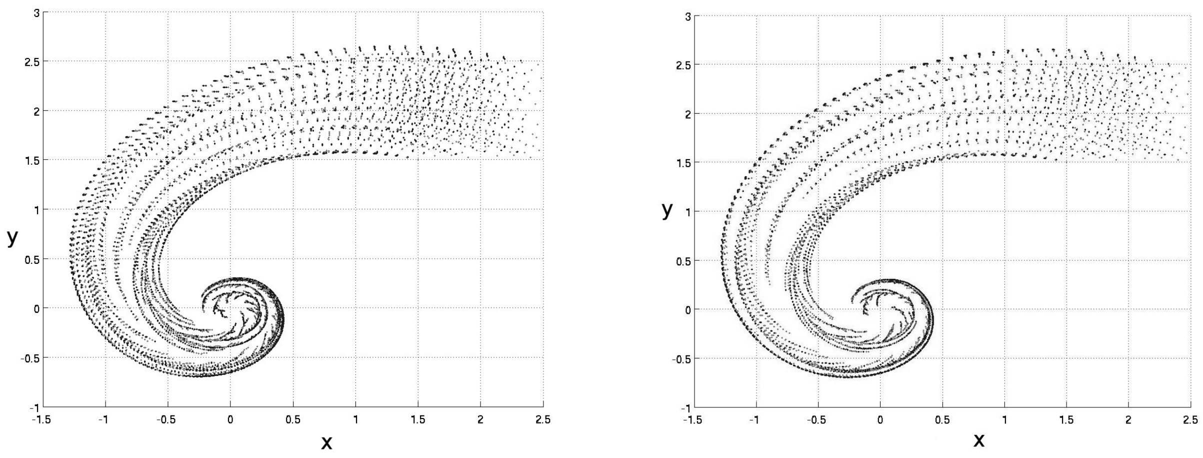

Figure 1.25: Results obtained using gRRT (left) and hRRT (right), with the same number of visited states.

These algorithms were implemented in the prototype tool HTG, which was successfully applied to treat a number of benchmarks in control applications and in analog and mixed-signal circuits [79, 31].

1.8 Conclusions

Embedded systems consist of hardware and software embedded in a physical environment with continuous dynamics. To model such systems, timed and hybrid automata models have been developed and studied extensively in the past two decades. In this chapter we have reviewed the basics of these models and methods of exhaustive or partial verification, as well as testing for these models. We hope that our overview will motivate embedded system designers to use these models in their applications, and that they will find them useful.

Timed and hybrid automata are still an active field of research, and we refer the readers to the numerous papers published on these topics, in addition to those referenced in our bibliography section.

Acknowledgements. We would like to thank Eugene Asarin, Olivier Bournez, Saddek Bensalem, Antoine Girard, Moez Krichen, Oded Maler, Tarik Nahhal, Sergio Yovine and other colleagues for their collaborations and their contributions to the results presented in this article.

References

[1] N. Abed, S. Tripakis, and J-M. Vincent. Resource-Aware Verification Using Randomized Exploration of Large State Spaces. In *SPIN'08*, number 5156 in LNCS, 2008.

[2] K. Altisen and S. Tripakis. Implementation of Timed Automata: an Issue of Semantics or Modeling? In P. Pettersson and W. Yi, editors, *3rd Intl. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS'05)*, volume 3829 of *LNCS*, pages 273–288. Springer, September 2005.

[3] R. Alur. Timed automata. NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems, 1998.

[4] R. Alur, C. Courcoubetis, N. Halbwachs, D.L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *3rd Conference on Concurrency Theory CONCUR '92*, volume 630 of *LNCS*, pages 340–354. Springer-Verlag, 1992.

[5] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[6] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[7] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivan, C. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems, 2002.

[8] R. Alur, T. Dang, and F. Ivancic. Counter-example guided predicate abstraction of hybrid systems. *Theoretical Computer Science (TCS)*, 354(2):250–271, 2006.

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

[10] H. Anai and V. Weispfenning. Reach set computations using real quantifier elimination. In M.D. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control*, LNCS 2034, pages 63–75. Springer-Verlag, 2001.

[11] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In B. Krogh and N. Lynch, editors, *Hybrid Systems: Computation and Control*, LNCS 1790, pages 20–31. Springer-Verlag, 2000.

[12] E. Asarin, T. Dang, and A. Girard. Hybridization methods for the analysis of nonlinear systems. *Acta Informatica.*, 43(7):451–476, 2007.

[13] E. Asarin, T. Dang, and O. Maler. The d/dt tool for verification of hybrid systems. In *Computer Aided Verification*, LNCS 2404, pages 365–370. Springer-Verlag, 2002.

[14] E. Asarin and G. Schneider. Widening the boundary between decidable and undecidable hybrid systems. In *CONCUR*, 2002.

[15] J. Beck and W. W. L. Chen. Irregularities of distribution. In *Acta Arithmetica*, UK, 1997. Cambridge University Press.

[16] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. *IFIP Congress Series*, 9:41–46, 1983.

[17] A. Bhatia and E. Frazzoli. Incremental search methods for reachability analysis of continuous and hybrid systems. In *HSCC*, pages 142–156, 2004.

[18] S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In W.P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference, Intl. Symposium (COMPOS'97)*, volume 1536 of *LNCS*, pages 103–129. Springer, September 1998.

[19] D. Bosnacki. Digitization of timed automata. In *Proc. of the Fourth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '99)*, pages 283–302, 1999.

[20] O. Botchkarev and S. Tripakis. Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In B. Krogh and N. Lynch, editors, *Hybrid*

Systems: Computation and Control, LNCS 1790, pages 73–88. Springer-Verlag, 2000.

[21] O. Bournez, O. Maler, and A. Pnueli. Orthogonal polyhedra: Representation and computation. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control*, LNCS 1569, pages 46–60. Springer-Verlag, 1999.

[22] P. Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3):281–320, 2004.

[23] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Are timed automata updatable? In *CAV'00*. LNCS 1855, 2000.

[24] M. Bozga, O. Maler, and S. Tripakis. Efficient Verification of Timed Automata using Dense and Discrete Time Semantics. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference (CHARME '99)*, volume 1703 of *LNCS*, pages 125–141. Springer, September 1999.

[25] M. Branicky, M. Curtiss, J. Levine, and S. Morgan. Sampling-based reachability algorithms for control and verification of complex systems. In *Thirteenth Yale Workshop on Adaptive and Learning Systems*, 2005.

[26] K. Cerans and J. Viksna. Deciding reachability for planar multi-polynomial systems. In *Hybrid Systems*, pages 389–400, 1995.

[27] A. Chutinan and B.H. Krogh. Verification of polyhedral invariant hybrid automata using polygonal flow pipe approximations. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control*, LNCS 1569, pages 76–90. Springer-Verlag, 1999.

[28] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. Journal of Foundations of Computer Science*, 14(4):583–604, 2003.

[29] T. Dang. Reachability-based technique for idle speed control synthesis. *International Journal of Software Engineering and Knowledge Engineering IJSEKE*, 15 (2), 2005.

[30] T. Dang and O. Maler. Reachability analysis via face lifting. In T.A. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, LNCS 1386, pages 96–109. Springer-Verlag, 1998.

[31] T. Dang and T. Nahhal. Using disparity to enhance test generation for hybrid systems. In *TESTCOM/FATES*, LNCS. Springer, 2008.

[32] T. Dang and T. Nahhal. Model-based testing of hybrid systems. Technical report, Verimag, IMAG, Nov 2007.

[33] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The Tool KRONOS. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III: Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer, 1996.

[34] C. Daws and S. Tripakis. Model Checking of Real-Time Reachability Properties Using Abstractions. In B. Steffen, editor, *4th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 313–329. Springer, 1998.

[35] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.

[36] A. Donzé and O. Maler. Systematic simulation using sensitivity analysis. In *HSCC*, pages 174–189, 2007.

[37] J. Esposito, J. W. Kim, and V. Kumar. Adaptive RRTs for validating hybrid robotic control systems. In *Proceedings Workshop on Algorithmic Foundations of Robotics*, Zeist, The Netherlands, July 2004.

[38] J.C. Fernandez, C. Jard, T. Jéron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96*, volume 1102 of *LNCS*. Springer, 1996.

[39] G. Frehse, B. Krogh, R. Rutenbar, and O. Maler. Time domain verification of oscillator

circuit properties. *Electr. Notes Theor. Comput. Sci.*, 153(3):9–22, 2006.

[40] A. Girard. Reachability of uncertain linear systems using zonotopes. In *Hybrid Systems: Computation and Control*, LNCS 3414, pages 291–305. Springer, 2005.

[41] A. Girard and C. Le Guernic. Zonotope/hyperplane intersection for hybrid systems reachability analysis. In *Hybrid Systems: Computation and ControlHSCC*. Springer, 2008.

[42] A. Girard, C. Le Guernic, and O. Maler. Efficient computation of reachable sets of linear time-invariant systems with inputs. In *Hybrid Systems: Computation and ControlHSCC*, pages 257–271. Springer, 2006.

[43] A. Girard and G. Pappas. Verification using simulation. In *HSCC*, pages 272–286, 2006.

[44] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *SIGPLAN Not. (PLDI'05)*, 40(6):213–223, 2005.

[45] M.R. Greenstreet and I. Mitchell. Reachability analysis using polygonal projections. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control*, LNCS 1569, pages 76–90. Springer-Verlag, 1999.

[46] R. Grosu, X. Huang, S.A. Smolka, W. Tan, and S. Tripakis. Deep Random Search for Efficient Model Checking of Timed Automata. In F. Kordon and O. Sokolsky, editors, *7th Monterey Workshop on Composition of Embedded Systems*, volume 4888 of *LNCS*. Springer, October 2006.

[47] T. Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Journal of Computer and System Sciences*, pages 373–382. ACM Press, 1995.

[48] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP'92*, LNCS 623, 1992.

[49] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[50] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[51] G. Holzmann. An analysis of bitstate hashing. In *Formal Methods in System Design*, pages 301–314. Chapman & Hall, 1998.

[52] G. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.

[53] S. Iman and S. Joshi. *The e-Hardware Verification Language*. Springer, 2004.

[54] C. Jard and T. Jeron. Bounded-memory Algorithms for Verification On-the-fly. In *CAV'91*, volume 575 of *LNCS*. Springer, 1992.

[55] A. Agung Julius, Georgios E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust test generation and coverage for hybrid systems. In *HSCC*, pages 329–342, 2007.

[56] J. Kapinski, B. Krogh, O. Maler, and O. Stursberg. On systematic simulation of open continuous systems. In *HSCC*, pages 283–297, 2003.

[57] J. Kim, J. Esposito, and V. Kumar. Sampling-based algorithm for testing and validating robot controllers. *Int. J. Rob. Res.*, 25(12):1257–1272, 2006.

[58] Donald E. Kirk. *Planning Algorithms*. Dover Publications, April 2004.

[59] M. Kloetzer and C. Belta. Reachability analysis of multi-affine systems. In *Hybrid Systems: Computation and Control*, pages 348–362. Springer, 2006.

[60] M. Krichen and S. Tripakis. Conformance Testing for Real-Time Systems. Formal Methods in System Design (to appear).

[61] M. Krichen and S. Tripakis. Black-Box Conformance Testing for Real-Time Systems. In S. Graf and L. Mounier, editors, *11th Intl. SPIN Workshop on Model Checking Software (SPIN'04)*, volume 2989 of *LNCS*, pages 109–126. Springer, April 2004.

[62] M. Krichen and S. Tripakis. Real-time Testing with Timed Automata Testers and Coverage Criteria. In Y. Lakhnech and S. Yovine, editors, *Joint Intl. Conf. on Formal*

Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault-Tolerant Systems, FORMATS/FTRTFT 2004, volume 3253 of *LNCS*, pages 134–151. Springer, September 2004.

[63] M. Krichen and S. Tripakis. State Identification Problems for Timed Automata. In F. Khendek and R. Dssouli, editors, *17th IFIP TC6/WG 6.1 Intl. Conf. on Testing of Communicating Systems (TestCom'05)*, volume 3502 of *LNCS*, pages 175–191. Springer, May 2005.

[64] A. Kuehlmann, K. McMillan, and R. Brayton. Probabilistic state space search. In *ICCAD'99*, pages 574–579, 1999.

[65] J. Kuffner and S. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA'2000), San Francisco, CA*, April 2000.

[66] A. Kurzhanski and I. Valyi. *Ellipsoidal Calculus for Estimation and Control.* Birkhauser, 1997.

[67] A.B. Kurzhanski and P. Varaiya. Ellipsoidal techniques for reachability analysis. In *Hybrid Systems: Computation and Control*, 2000.

[68] A. A. Kurzhanskiy and P. Varaiya. Ellipsoidal toolbox (et). In *Proc. 45th IEEE Conf. on Decision and Control*, 2006.

[69] M. Kvasnica, P. Grieder, M. Baoti, and M. Morari. Multi-parametric toolbox (mpt). In *Hybrid Systems: Computation and Control*, volume LNCS 2993, pages 448–462. Springer, 2004.

[70] K. Larsen, P. Petterson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1/2), October 1997.

[71] S. LaValle and J. Kuffner. Rapidly-exploring random trees: Progress and prospects, 2000. In Workshop on the Algorithmic Foundations of Robotics.

[72] Steve LaValle. *Planning Algorithms.* Cambridge University Press, 2006.

[73] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84:1090–1126, 1996.

[74] J. Lygeros, K. Johansson, S. Sastry, and M. Egerstedt. the existence of executions of hybrid automata. In *in IEEE Conference on Decision and Control, Phoenix, AZ*, 1999.

[75] M. Mihail and C. H. Papadimitriou. On the random walk method for protocol testing. In David L. Dill, editor, *Proc. 6th Intl. Conf. on Computer-Aided Verification CAV*, volume 818, pages 132–141. Springer, 1994.

[76] O. Maler and A. Pnueli. Reachability analysis of planar multilinear systems. In *Proceedings of the 4th Computer-Aided Verification*, volume 697. Springer, 1993.

[77] Ian M. Mitchell and Jeremy A. Templeton. A toolbox of Hamilton-Jacobi solvers for analysis of nondeterministic continuous and hybrid systems. In *Hybrid Systems: Computation and Control*, LNCS. Springer-Verlag, 2005, to appear.

[78] N. Kitchen and A. Kuehlmann. Stimulus generation for constrained random simulation. In *ICCAD 2007*, pages 258–265, 2007.

[79] T. Nahhal and T. Dang. Test coverage for continuous and hybrid systems. In *CAV*, pages 454–468, 2007.

[80] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In *Hybrid Systems*, pages 149–178, 1992.

[81] J. Ouaknine and J. Worrell. Revisiting digitization, robustness, and decidability for timed automata. In *LICS 2003.* IEEE CS Press, 2003.

[82] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6), 1987.

[83] G. Pappas, G. Lafferriere, and S. Yovine. A new class of decidable hybrid systems. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control*, LNCS 1569, pages 29–31. Springer-Verlag, 1999.

[84] R. Pelanek and I. Cerna. Enhancing random walk state space exploration. In *In Proc. of Formal Methods for Industrial Critical Systems (FMICS'05)*, pages 98–105. ACM Press, 2005.

[85] E. Plaku, L. Kavraki, and M. Vardi. Hybrid systems: From verification to falsification. In W. Damm and H. Hermanns, editors, *International Conference on Computer Aided Verification (CAV)*, volume 4590, pages 468–481. Lecture Notes in Computer Science, Springer-Verlag Heidelberg, Berlin, Germany, 2007.

[86] S. Prajna and A. Jadbabaie. Safety verification of hybrid systems using barrier certificates. In Rajeev Alur and George J. Pappas, editors, *Hybrid Systems: Computation and Control*, volume 2993 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2004.

[87] S. Prajna, A. Papachristodoulou, P. Seiler, and P. A. Parrilo. *SOSTOOLS: Sum of squares optimization toolbox for MATLAB*, 2004.

[88] A. Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.

[89] A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 95–104, Stanford, California, USA, 1994. Springer-Verlag.

[90] Stefan Ratschan and Zhikun She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *ACM Trans. Embedded Comput. Syst.*, 6(1), 2007.

[91] S. Sankaranarayanan, T. Dang, and F. Ivancic. Symbolic model checking of hybrid systems using template polyhedra. In *TACAS'08 - Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[92] S. Shyam and V. Bertacco. Distance-guided hybrid verification with GUIDO. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1211–1216. European Design and Automation Association, 2006.

[93] J. Sifakis and S. Yovine. Compositional specification of timed systems. In *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, volume 1046 of *LNCS*. Spinger-Verlag, 1996.

[94] O. Stursberg and B. Krogh. Efficient representation and computation of reachable sets for hybrid systems. In *Hybrid Systems: Computation and Control HSCC*, volume LNCS, pages 482–497. Springer, 2003.

[95] L. Tan, J. Kim, O. Sokolsky, and I. Lee. Model-based testing and monitoring for hybrid embedded systems. In *proceedings of IEEE Internation Conference on Information Reuse and Integration (IRI'04)*, 2004.

[96] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002.

[97] A. Tiwari and G. Khanna. Nonlinear systems: Approximating reach sets. In *Hybrid Systems: Computation and Control*, volume 2993 of *Lecture Notes in Computer Science*, pages 600–614. Springer, 2004.

[98] C. Tomlin, I. Mitchell, A. Bayen, and M. Oishi. Computational techniques for the verification of hybrid systems. *Proceedings of the IEEE*, 91(7):986–1001, 2003.

[99] F. Torrisi and A. Bemporad. HYSDEL - a tool for generating computational hybrid models. *IEEE Transactions on Control Systems Technology*, 12(2):235–249, 2004.

[100] J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99*, volume 1664 of *LNCS*. Springer, 1999.

[101] S. Tripakis. Checking Timed Büchi Automata Emptiness on Simulation Graphs. ACM Transactions on Computational Logic (to appear).

[102] S. Tripakis. Fault Diagnosis for Timed Automata. In W. Damm and E.-R. Olderog, editors, *Formal Techniques in Real Time and Fault Tolerant Systems, 7th Intl. Sym-*

posium (FTRTFT'02), volume 2469 of *LNCS*, pages 205–224. Springer, September 2002.

[103] S. Tripakis. Folk Theorems on the Determinization and Minimization of Timed Automata. *Information Processing Letters*, 99(6):222–226, September 2006.

[104] S. Tripakis. What is Resource-Aware Verification? Unpublished document, 2008. Available from the author's web page.

[105] S. Tripakis and C. Courcoubetis. Extending Promela and Spin for Real Time. In T. Margaria and B. Steffen, editors, *2nd Intl. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 329–348. Springer, March 1996.

[106] S. Tripakis and S. Yovine. Analysis of Timed Systems using Time-abstracting Bisimulations. *Formal Methods in System Design*, 18(1):25–68, January 2001.

[107] A. van der Schaft and H. Schumacher. *An introduction to hybrid dynamical systems.* LNCIS 251, Springer, Berlin, 2000.

[108] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification.* Elsevier, 2005.

[109] M. De Wulf, L. Doyen, and J-F. Raskin. Almost ASAP semantics: From timed models to timed implementations. In *Hybrid Systems: Computation and Control (HSCC'04)*, volume 2993 of *LNCS*. Springer, 2004.

[110] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems. In *5th Intl. Conf. on Computer-Aided Verification*, LNCS 697, June 1993.

[111] J. Yuan, C. Pixley, and A. Aziz. *Constraint-Based Verification.* Springer, 2006.

[112] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4), 1997.