The Unmet Challenge of Timed Systems

Oded Maler

CNRS-VERIMAG, University of Grenoble
2, av. de Vignate, 38610 Gieres, France
@imag.fr

Dedicated to Joseph Sifakis.

Abstract. *Timed systems* constitute a class of dynamical systems that live in an extremely useful level of abstraction. The paper stresses their importance in modeling without necessarily endorsing the orthodox approach for reasoning about them which is practiced in the theoretical and applied branches of formal verification.

1 Introduction

This paper is about *timed systems*, a formal model that combines discrete transitions and metric time. Joseph has been involved in studying such systems during several periods of his career including work on timed Petri nets, timed process algebra and, more effectively, in the context of algorithmic formal verification where he (together with students and collaborators) played an important role in the evolution of timed automata modeling and verification [18, 30, 15, 12, 5, 27]. More recently, as a promoter of a compositional approach to embedded systems design [8, 7], I guess he could observe how real-time and performance tend to pop-up and demonstrate yet another difference between a nice theory and practical reality. Performance is a non-compositional phenomenon per se because it involves sharing of limited resources and the performance of a single process in isolation typically degrades when it has to share resources with others.

Since I spent significant parts of my academic life working on timed systems at Joseph's VERIMAG, I find it appropriate to use this opportunity to reflect aloud on this topic, free from the usual ceremony of theorems, tools and case-studies found in standard publications. A large part of this paper will be situated on the abstract and meta level, but I will start by formulating a concrete motivating[1] problem more related to the expected audience.

Consider an application program such as a video decoder to be executed on a new mobile multi-core platform. The application is structured as a *task graph*, a collection of tasks,[2] partially-ordered according to precedence, and annotated by execution times and data transfer rates. We assume that these durations, as well as the arrival rates of new jobs to execute, admit some *bounded uncertainty* which can be due to different factors

[1] The term "motivating" is used in a very liberal and wide sense, covering psychological, sociological and metabolic aspects of scientific activity.

[2] Modules, components, actors, filters, functional blocks, stream transducers...

ranging from variability in the type of image frames and the workload of the platform, down to the possibility that the software and/or hardware are still in the design stage. Executing the program on the architecture, in addition to the usual compilation chain, involves other orchestration decisions for which we use the collective term *deployment*: how to map tasks to processors, how to schedule them, how to allocate buffers and communication channels and how to transfer data among processors. All these questions involve non-trivial combinatorial problems and bad choices can have far reaching effects on performance to the point of making the difference between the feasible and the infeasible. Forcing application programmers to occupy themselves with these issues is like returning to the stone age of computing, undoing the impressive progress made over the years by isolating programmers from more and more low-level implementation details. Hence it is urgent to provide computer-aided support for multi-core deployment. Timed systems provide, in principle, the conceptual and mathematical foundations for evaluating, comparing and optimizing such deployment decisions.

The message of this paper can be summarized as follows. Models of timed systems are extremely important as they represent a level of abstraction which is used, explicitly or implicitly, in almost any domain of engineering and everyday life. In particular, timed models have a significant potential contribution to performance analysis and optimization for all sorts of systems. Unfortunately, sociological and cultural factors, both in academia and industry, as well as genuine complexity problems, prevent this potential from being fully realized.

The rest of the paper is organized as follows. Section 2 presents the timed level of abstraction and demonstrates how it can be used in modeling. Timed automata are introduced in Section 3 along with their non-scalable orthodox analysis in the tradition of formal verification and various attempts to make it scale up. Section 4 recounts some recent encounters with practice and provides retrospective reflections on timed and untimed verification without a decisive punch line. Anecdotes and strong opinions, not always well-founded, are interleaved in the text and should not be taken too seriously and certainly not offensively.

2 The Timed Level of Abstraction

It is a common knowledge that phenomena can be modeled at different levels of abstraction or granularity. Lower levels are more detailed, zoomed at more elementary entities and are supposed to be closer to *deep reality*.[3] I will start by discussing abstractions based on *aggregation* which are more common in Physics and its Applied Mathematics (and also in macro Economics) and then move to other types of abstractions, more relevant to our concerns.

The prime example of aggregation-type abstraction is found in Statistical Physics where the underlying detailed micro model consists of a huge number of particles whose respective momenta are aggregated and abstracted into coarse-grained macro entities such as temperature. The derivation of the macro model from the micro-level rules

[3] This is at least what reductionists (physicists, molecular biologists and even humble programmers) want us to believe.

often exists only in principle. My favorite example[4] is the *module of elasticity* which indicates the resistance of a beam to different loads and serves for reasoning about building stability. This characterization is inferred from macro level experiments on beams made of various materials and is *not* derived from a detailed model of zillions of interacting molecules.

When models are more detailed, you have to deal with more state variables and this raises two main problems. In the interface between the real world and the model you face the problem of observation/measurement: you need to keep track of many particles in order to identify the system dynamics and also to measure initial conditions if you want to use the model to predict the future. Even if this scientific model building and measurement problems are somehow miraculously solved and you have a model, you still have a problem in the virtual world of ideas and numbers because it takes much more effort to do *computations* with the model. This difficulty applies both to the (increasingly more rare) cases where you can apply analytic/symbolic methods, as well as to the modern way to reason about complex systems, namely, *simulation*. This computational difficulty applies even more to *algorithmic verification*, also known as *model-checking*, which is essentially a kind of extended muscular form of simulation, augmented sometimes with a more expressive temporal language for evaluating and classifying behaviors [22].

Another type of abstraction, perhaps more familiar to people from our close domains, is *model reduction* which eliminates some variables and simplifies the dynamics but remains within the same order of magnitude in terms of the number of variables. Such reductions are common in the treatment of models of continuous dynamical systems defined by ordinary differential equations (ODE) in Control and related engineering disciplines but also in discrete models where some variables are hidden. The formal relationship between the original and reduced models is based on the projection of the behaviors of both systems on common observable variables. For ODE models, the reduced system should produce behaviors (trajectories) which are *close* to the detailed one in some metric. In the Computer Science tradition, projecting away variables renders the system non-deterministic in a set-theoretic sense, first introduced in [26].[5] Hence the abstract system may have several behaviors emanating from the same initial state and the relation between the two levels is typically a *conservative approximation*: the set of behaviors of the abstract model is a superset of those of the concrete system.

In the compilation of high-level programs into machine code the relation between different levels is based on a combination of variable hiding and a change in the time scale. A program instruction which is considered atomic at high-level corresponds to a more detailed sequence of primitive instructions involving addressing and registers which are hidden from the high-level models. Proceeding downward, each of the machine code instructions is refined into a collection micro-code hardware transactions and so on. Going upwards, pieces of program code are grouped into *procedures* that can be used, in their turn, as atomic instructions. It is worth noting that software and

[4] Dedicated to my father, Ephraim Maler, a construction engineer.

[5] The adaptation to continuous and hybrid systems is elaborated in [23] where such systems are referred to as *under-determined* to avoid the term *non-deterministic* which is already overloaded with connotations.

digital hardware are remarkably exceptional in the sense of having a *formal equivalence* between so many levels. This is due to the aphysical (and sequential) nature of computational models, the underlying hardware infra-structure and the fact that we deal here with man-made artifacts rather than mother nature.

Finally there is a class of abstractions which transform a systems defined over real-valued numerical variables into a discrete-event system where ranges of continuous variables are quantized into abstract states. Two of the concrete examples given below in order to introduce the idea of timed systems are based on such a quantization while the third is taken from the software domain.

Example 1: Transistors and Gates. At a lower-level a logical gate, say inverter, is an electrical circuit composed of transistors whose voltage at the output port responds to the voltage in its input. Its behavior is a signal, a trajectory of a continuous dynamical system which has two stable states around voltages v_0 and v_1. At the abstract Boolean level we say that when the input goes down, the gate becomes excited and takes an observable transition from 0 to 1 as shown in Fig. 1-(a).

Example 2: Coming to Grenoble. Suppose you come to Grenoble from your home-town by airplane via Lyon airport. A low-level description can be given in terms of the trajectory of your center of mass on the spatial Earth coordinates. At a higher level you can describe it verbally as a sequence of events: fly to Lyon then wait for the bus that will take you to Grenoble, see Fig. 1-(b).

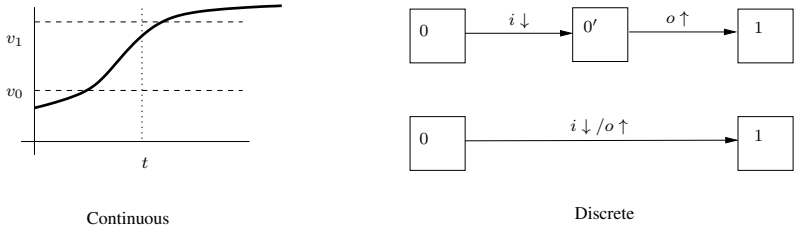
Example 3: Software. At a lower level we have piece of code, an algorithm that transforms some input to some output using instructions that run on some hardware platform. At a higher level of description we say that we process an image frame, e.g. decode or filter it (Fig. 1-(c)).

From the more abstract discrete point of view, in all these cases you have some kind of a (concrete, physical) process that you do not care too much about its *intermediate* states.The important thing is that at the end of the process you will be in Grenoble, the gate will switch properly from 0 to 1 according to Boole-Shannon rules and the image will be decoded. This is the essence of functional reasoning. But you cannot get rid completely of the underlying Physics.[6] In order to determine the clock rate that your computer can use, you need to know *how long* it takes to switch from 0 to 1.[7] To watch some stupid video on your smart phone you do care about the *execution time* of your decoding algorithm. To come on time to a conference you need to know the *duration* of the flight. The purely discrete automaton model does not distinguish between flying from Paris and flying from San Francisco: the flight is modeled simply as an abstract sequence of transitions: take-off $\rightarrow$ fly $\rightarrow$ land.

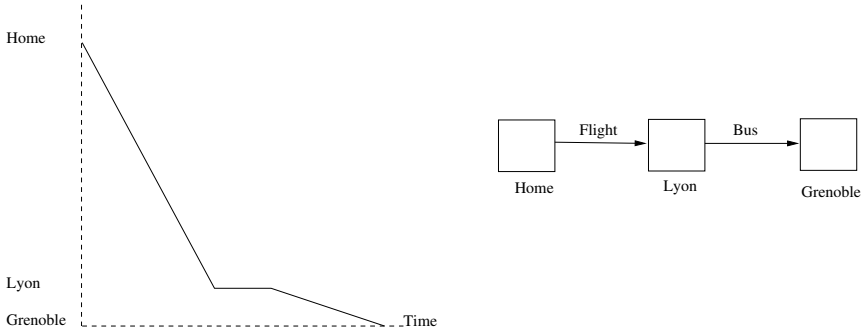
The timed level of abstraction offers a *compromise* between these two extremes, the fully continuous and the purely discrete. To understand what is going on, let us look closer at the nature of the discrete models and their relationships with the underlying continuous process. Such models observe the continuous variables through an abstraction/quanization: rather than recording the exact liquid level in your glass, it classifies

[6] The term "physics" is used here in a very loose sense as denoting all those quantitative things that preceded the abstract computer.

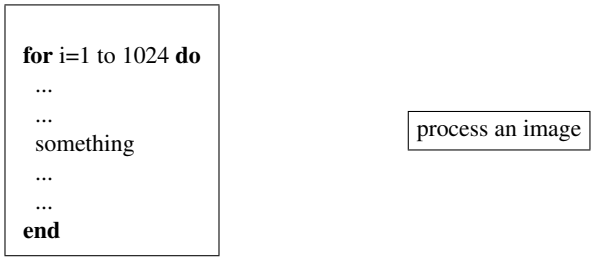
[7] In the development of digital circuits there is a phase called *timing closure* where these considerations, neglected in preceding design stages, are considered.



(a)



(b)



(c)

Fig. 1. Three examples of systems viewed in two adjacent levels of abstraction. (a) *Logical gate*. Left: a reaction of an inverter to a change in its input voltage; Right: The discrete model where from a stable state 0, the gate becomes excited (state $0'$) when its input goes down and then stabilizes into a new state 1. In some modeling styles the intermediate unstable state is ignored and the changes in input and output are part of the same transition. (b) *Coming to Grenoble*. Left: a simplified evolution of the distance from Grenoble during flight, waiting at the airport and taking the bus. Right: the trip viewed as a discrete sequence of steps. (c) A program and its abstract description.

the states into a finite number of categories, say, empty, full and in-between. Likewise, the *dynamics* of the water level can be classified as filling, emptying or static, which can be viewed as quantizing the derivative. Discrete events indicate changes in states which correspond to threshold crossings of continuous variables as well as changes in the dynamics such as opening and closing valves. Timed systems augment the expressive power of the discrete model with a *metric temporal distance* between events or the *duration* of staying in states. As mathematical objects, timed behaviors are represented in two fundamental forms that correspond to the two types of timed monoids described in [4]:

- The first representation is based on *time-event sequences* where a behavior is viewed as an alternation of durations and instantaneous events. This is essentially equivalent to the timed traces used in [3] in which the events are embedded in the real-time axis and represented as sequences of time-stamped events. Such representations are produced in numerous domains including all sorts of event monitoring systems as well as numerical simulators for differential equations.
- The second representation is based on discrete-valued *signals* which are functions from real time to finite domains. Boolean signals are heavily used in the design of digital circuits. Similar mathematical objects are used in all planning and scheduling domains where they are called Gantt charts or time-tables.

Fig. 2 depicts timed descriptions of the trip to Grenoble in these two forms. Using such a representation we can distinguish short flights from long, fast gates from slow and efficient algorithms from less efficient ones.

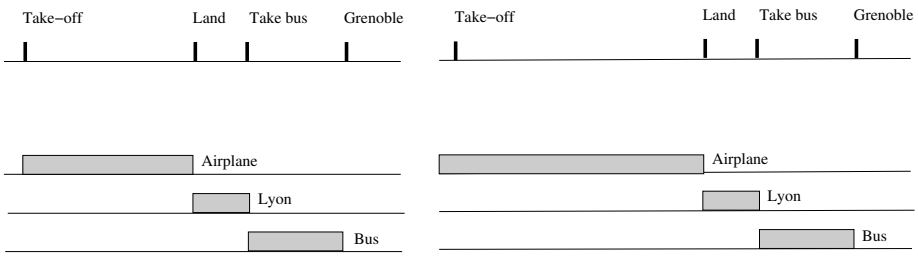


Fig. 2. The trip viewed as at a timed level of abstraction: as a sequence of events embedded in the real time axis (up) or as signals over discrete domains (down). At this level of abstraction one can tell the difference between a short flight (left) and a longer one (right).

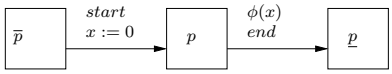
This gives rise to a new class of dynamical systems which, unfortunately, is a bit hard to digest. Recall that continuous behaviors are trajectories in a continuous state-space of dynamical systems specified using *differential equations* and that models based on *automata* are the dynamical systems that produce discrete behaviors, sequences of states and events. Likewise *timed automata* and similar timed models generate timed behaviors. These are the *dynamical systems of the timed level of abstraction.*

The basic atomic component of the timed world is the process that takes some time to conclude after having started. Such a process is modeled by the simple timed automaton depicted in Fig. 3-(a). It consists of an *idle* state $\overline{p}$ in which nothing happens and from which a *start* transition takes it to *active* state p while resetting clock x to zero. Clock x is a restricted type of state variable which progresses in the speed of time ($\dot{x} = 1$) so as to measure the time elapsed since an activity has started. The influence of x on the dynamics is via its participation in the precondition (the transition guard ϕ) for taking the *end* transition to *final* state $\underline{p}$. The condition ϕ can be deterministic, $x = d$, indicating that process duration is assumed to be precisely known. It can be nondeterministic, $x \in [a, b]$, meaning that the duration can be anywhere within the interval. The degenerate case where the condition is always true, that is, $x \in [0, \infty)$, is equivalent to an untimed model. The condition can also be made probabilistic, assuming some distribution on the duration, resulting in an expressively rich kind of a continuous-time stochastic process [24].

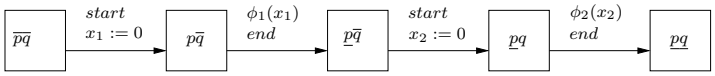
To my mind, the most important contribution of the theory of automata to humankind is in the notion of parallel composition (or products of automata). Such notions exist of course also for interacting continuous dynamical systems but only in automata you can visualize a global system and see the effect of interaction. When you compose two automata, you obtain a global automaton whose states are of the form $s = (s_1, s_2)$, elements of the Cartesian product of the individual state-spaces. Transitions available from s are either independent transitions taken from s_1 or s_2 or transitions of one automaton which are conditioned on the state of the other. Fig. 3-(b) shows how parallel composition can realize sequential composition: the *start* transition of the automaton of process q is conditioned the by the automaton for process p being in its final state. Sequential composition represents *precedence* relations between tasks where $p \prec q$ indicates that q cannot start before p has terminated. This is how you express statements like, *you can take the bus after you land*, *a gate switching triggers a change in the next gate* or *you can start processing the image only after having decoded it*.

More generally, parallel composition can express processes that run concurrently, sometimes progressing independently and sometimes synchronizing. In timed models, independent progress is not as independent because Time is viewed as a shared variable that all processes interact with. The automaton of Fig. 3-(c) shows a fragment of the composition of automata for two timed processes. In state s we observe the fundamental phenomenon in timing analysis: two or more active processes running in parallel.[8] The winner in this *race*, a term used in continuous-time stochastic processes, is the one which takes its *end* transition first. The identity of the winner depends on the delay bounds $[a_1, b_1]$ and $[a_2, b_2]$ as well as the timing of the preceding transitions. Typically, timing constraints, due to sharing of the time variable, restrict the range of behaviors which would be otherwise possible in the untimed transition graph. Analyzing the possible behaviors of such concurrent timed processes is at the heart of almost anything we do when we hurry up to catch a bus or wait for our partner to come. The following list illustrates the universality of questions related to possible behaviors (paths) of networks of timed automata.

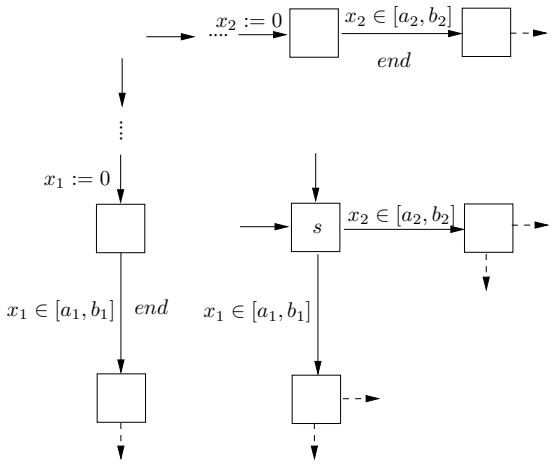
[8] For this reason I find papers that deal with timed automata with a *single* clock to be of a purely theoretical interest.



(a)



(b)



(c)

Fig. 3. Processes that take time: (a) The basic timed component; (b) Sequential composition; (c) Parallel composition. State s admits a race between two processes.

– Will there be a glitch in the circuit?
– Will he finish his boring talk by the coffee break?
– Will the meal be ready exactly when the guests arrive?
– Will my student finish the thesis before I run out of money?
– Will the image be processed before the arrival of the next one?
– Will the server answer the query before the attention span of the client expires?

I hope you are convinced by now that timed systems are important for *modeling* and you can use them to formulate all kinds of interesting questions in an extremely important level of abstraction. It is the level of abstraction that people use implicitly in scheduling, timing analysis, planning - you name it. It should be noted that timed automata (and other forms of timed transition systems such as timed Petri nets) are not unique in addressing this level of abstraction. For example, the work on Operations Research that involves scheduling deals with this level but it often jumps directly to a constrained optimization problem without passing through a dynamic model. Another example would be Queuing Theory which uses continuous-time stochastic processes that generate (distributions over) timed behaviors, but there, at least for a point of view of an innocent outsider, it seems that the heavy mathematical technology associated with probabilities over the reals dominates this work at the expense of the semantic view.[9] A dynamical system view of timed systems has been developed in the study of discrete-event system in Control [14] and also in the context of Max Plus algebra [6] but in the latter, due to linearity, the expressive power is rather limited.

The questions that remains is how can these timed models be used to provide *answers* to those interesting questions. To answer this particular question, let us have a retrospective look at *formal verification*, our home discipline.

3 Verification and Analysis of Timed Systems

A large part of verification is concerned with showing that components in a network of automata interact properly with each other. The term "properly" means that some sequences of events are considered acceptable while others violate the requirements. Violation means either that bad things happen, for example, two processes write simultaneously on a shared resource or an airplane crashes. Technically, such *safety* properties are violated by reaching an undesired part of the state-space. The other types of properties are called *liveness* properties and are violated when some good things do not happen, for example a client is starved to death without getting what he or she has requested.[10] The models used to verify such properties are discrete and often abstract away from data and focus on control/synchronization. The systems in question are open and under-determined and this means that a model will have *many* executions,

[9] I used to be a more zealous supporter of the semantic-dynamic approach [1] but like any other approach including those just mentioned, it has its advantages and shortcomings. Sometimes a semantically-correct "formal" approach stops at definitions and hardly computes anything.

[10] It is worth noting that for timed requirements, that is, when an upper-bound to an acceptable delay is specified, all properties can be viewed as safety properties [19].

some correct and some incorrect. Verification is a kind of *exhaustive simulation* which explores *all* the paths in a huge automaton.

Extending verification to timed systems means that, in addition to the under determination associated with external discrete actions, there is also a *dense* temporal nondeterminism concerning timing as we do not know execution times, propagation delays, inter-arrival times and process durations with precision but model them typically using *bounded intervals*. Following the pessimistic safety-critical spirit of verification, we attempt to reason *universally* about this uncertainty space, compute *all* possible behaviors under all choices of duration values and check the correctness of each of them. Without formal verification techniques this would amount to a non-countable number of simulations.

From the verification point of view we have a-priori a system with an infinite (and non-countable) state-space as the clocks are real-valued and the states of the timed automaton are of the form (s, x) with s being an automaton state and x a vector of clock valuations. Looking closer we can observe that clock valuations range practically over a bounded subset of $\mathbb{R}^n$ (clock values that go beyond the largest constant in the timing constraints are not interesting) and the restricted use of clocks in transition guards induces a finite-quotient property. More precisely, there is an equivalence relation $\sim$ over the set X of clock valuations such that $x \sim x'$ implies that the same sequences of transitions are possible from (s, x) and (s, x'). The relation is of finite index and hence a timed automaton is equivalent to a finite-state automaton with states of the from (s, R) where R is an equivalence class of $\sim$, also known as a *region*, and transitions correspond either to the passage of time from one region to another or to discrete transitions. This region graph was introduced in the seminal paper of Alur and Dill [3] which put timed automata on the map, and was used to prove decidability of the basic verification problem. Beyond this theoretical use, the region automaton is completely impractical and is not used by any living verification tool. The region equivalence is unnecessarily fine and makes distinctions between clock valuations that differ only by durations of time steps, but still admit the same sequences of transitions.

The other approach which uses a coarser equivalence has several origins [18, 28, 29] and in its contemporary form it computes a finite-state automaton, the *reachability graph* also known as the *simulation graph*, on the fly in a manner similar to the algorithmic analysis of hybrid systems described in [2]. The symbolic states are of the form (s, Z) where Z is a set of clock valuations belonging to a restricted type of polyhedra called *zones*, definable by conjunctions of inequalities of the form $c_1 \leq x_i \leq c_2$ or $c_1 \leq x_i - x_j \leq c_2$. Such difference constraints are fundamental to all sorts of timing and scheduling problems and they admit an efficient representation using difference-bound matrices (DBMs) invented by Bellman and proposed in the verification context in [17]. The regions of the region graph are the smallest zones possible.

To avoid large n-tuples let me illustrate zone-based reachability computation on the 2-clock automaton of Fig. 4, giving priority to clarity over precision. It starts at state q_1 with $x = (0, 0)$. Then the time elapse operator is used to compute all states reachable by letting time pass where the staying condition (invariant) $x \leq 3$ restricts the clock values with which it is possible to stay in q_1. Then this set is intersected with the transition guard $x \geq 1$ to yield the clock values with which the transition to q_2 is possible. The

transition resets clock x_1 and hence the possible starting points at q_2 are $(0, y)$ where $y \in [1, 3]$. Then the time elapse operator is applied at q_2 and so on and so forth. The process is guaranteed to terminate and compute all state reachable by any choice of delay values. It has however an annoying non-intuitive feature that it shares also with verification of hybrid systems and which makes it hard to explain to potential clients of this technology. Unlike simulation of differential equations (and simulation in general) there is no simple relationship between the steps of the algorithm and the flow of time. For example, after the first transition we are in q_2 with a set of initial clock valuations $1 \leq y \leq 3$ such that each of them has been reached at a different absolute time $t = y$, and this becomes more complicated for automata having several transitions outgoing from the same state.

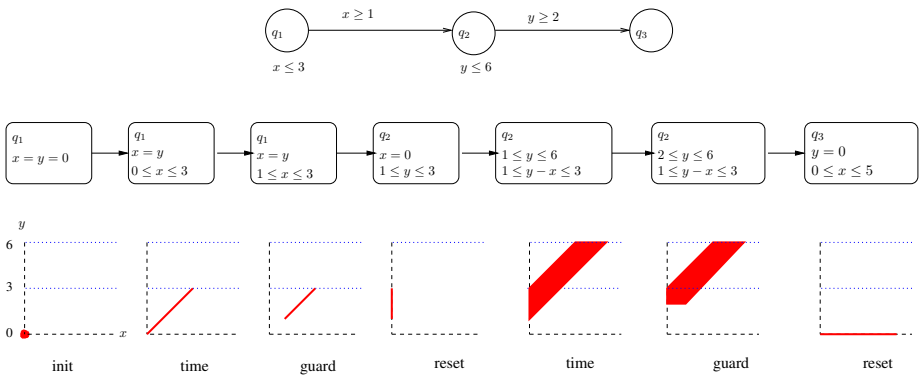


Fig. 4. First steps in computating a reachability graph.

It is worth giving here two small lessons in the methodology and history of science. The first is about the mathematician's obsession with being the *first* to prove a theorem. The finite quotient property of timed systems was described in [11] in the context of timed Petri nets, six years before the conference version of [3], and it already involved zones. This went mostly unnoticed and only the later exposure of the verification community, which was already working on real-time models [20], to these ideas created the impact. So it is not only the "result" that counts but also the language that you use, the attitude and capabilities of the community to which you present your work, the right timing and more. The story told in [16] about the discovery of planets is relevant.

The second lesson is about complexity: the size of the region graph can be exponential in the number of clocks while that of the reachability graph is potentially double-exponential, but in reality the latter is almost always smaller. So proving complexity bounds on this or that problem can sometimes be no more than a sterile exercise.

Verification algorithms have been implemented and improved in a series of theses and tools. At VERIMAG, under the guidance of Joseph, this lead to the tools Kronos, Open-Kronos and IF with contributions of Sergio Yovine, Alfredo Olivero, Conrado Daws, Stavros Tripakis and Marius Bozga. The most celebrated and actively maintained

tool these days is tool UPPAAL, started by Wang Yi, Paul Pettersen, and Kim Larsen as a collaboration between Uppsala and Aalborg and continued under the ongoing enthusiastic leadership of Kim with major contributions by G. Behrman and A. David. Despite enormous investments and some impressive achievements, I think it is fair to say that this approach rarely scales up beyond toy problems (and is also PSPACE-hard). Being convinced in the importance of timed systems for modeling and analysis I also spent around ten years of my life (and also those of students and collaborators) in trying to scale up and fight the clock explosion. Below is a short description of these attempts.

Numerical decision diagrams (NDD). One of the main problems in the verification of timed automata is the lack of a unified symbolic representation both for the discrete state-space and the clock-space. The representation is enumerative in the former and hence not suited for handling systems consisting of many components. NDDs give such a unified symbolic representation for discretized clocks encoded in binary. The technique worked well on one example but it had the deficiency of losing the metric structure of numbers via the binary encoding (known as bit blasting in the satisfiability jargon). Nevertheless it helped dispel some naive beliefs in the universal power of BDDs and clarify an important issue: dense time is *not* the main issue in timed automata but rather the symbolic representation of sets of clock valuations by inequalities.

Timed polyhedra. Another canonical representation for unions of zones was obtained as an extension of a canonical representation for orthogonal polyhedra. This was nice theoretically but at the end did not work because of lack of efficient representation of sets of permutations.

Heuristic search for scheduling. In scheduling under certainty (durations are known) you have a synthesis rather than verification problem and an optimal solution corresponds to the shortest path (in terms of duration) in a timed automaton. If you do not insist on optimality you need not be exhaustive. Moreover, under certain general conditions that apply, for example, to job-shop scheduling, there is a finite and discrete set of paths to consider and you need not handle zones. This does not solve the general verification problem, but similar ideas have been tried under the title of guided search, with the goal of finding bad behaviors quickly.

Bounded model-checking with SMT solvers. Bounded model-checking for timed automata, that is, the existence of a run with a bounded number of transitions, can be expressed by a formula in the logic of difference constraints. It turned out that even very strong solvers developed for this logic did not help in verifying timed automata and they even had a poorer performance than standard zone-based reachability.

Interleaving reduction One problem that adds to the high cost of zone-based reachability is that commuting paths do not really commute because the zone constraints remember the past history. Having shown that the union of zones reached by all interleavings of the same set of local transitions is convex, we developed a breadth-first reachability algorithm that merges such zones and for some period we held the olympic record in verifying Fisher's protocol.

Compositional timing analysis. This last heroic effort [10, 9] was based on a divide-and-conquer methodology applied to Boolean circuits with bi-bounded delays with each gate modeled as a timed automaton according the principles explained in [25]. The automata for a sub-circuit were analyzed and then abstracted by hiding internal clocks

and transition resulting in a small-size over-approximation which was plugged to the rest of the circuit. This technique could analyze a wave pipelining scheme for 3 waves of input to a non-trivial circuit of 36 gates, still a far cry from the needs of circuit timing analysis.

So was all this a waste of time? Before giving a hopefully negative answer let me reflect a bit on the state of science. Ideally one would like to apply noble first class science and mathematics to solve real problems. For example, Formal Language Theory and Compilation, Information Theory and Telecommunication, Number Theory and Cryptography. We accept good mathematics for its own sake as well as technological innovations produced by people who do not formulate themselves in a clean mathematical way. However, we should be careful not to commit the double sin of doing mediocre mathematics over marginal questions under the pretext of hypothetical applications, but this seems to be unavoidable in the current state of affairs and the structure of scientific communities (and industry). Of course, these defects are more easily detected on others than on oneself.

4 Strange Encounters with Reality

In the sequel I report some impressions from participating in an industrial-academic project where we promised to progress toward solving the multi-core deployment problem mentioned in the introduction. The self-confidence was based partly on our acquaintance with timed automata, scheduling and SMT solvers. Most of the observations are known to many people but each person discovers the facts of life in his own path, pace and order.

Between theoreticians and (real) practitioners. The theoretician has the liberty to choose the problems and ignore aspects which are outside the scope of his interest and his capabilities. The real[11] practitioner does not have this choice, his deadlines are not self-imposed and his time is measured. The theoretician solves *general* problems: verification applies, in principle, to all automata, all temporal logic formulae, etc. The practitioner solves *one* problem at a time. Consequently the real-life scope of a theoretical solution is any number of problems in $[0, \infty)$. It is zero if the compromise with reality was too aggressive, and infinity if it was a clever one. As a theoretician I can observe that $[0, \infty)$ and 1 are not comparable.

Correctness and Performance. I hold the view that correctness is a special (Boolean) case of a *performance measure*, which is a way to associate cost/utility with individual system behaviors and with the system as a whole. We can measure elapsed time, associate costs with states and transitions and accumulate them along runs. We need not necessarily Booleanize them via inequalities such as deadline conditions - we can remain *quantitative*. We should provide real numbers (and vectors of real numbers when we have multiple evaluation criteria) as answers. Many people will agree on that and performance evaluation is a major issue in the embedded world and elsewhere.

Who needs universal quantification? Due to safety-criticality or cost-criticality (hardware errors) verification always aspired to cover *all* possible points in the uncertainty

[11] The distinction between real and less real is due to Paul Caspi [13].

space, in other words, a pessimistic *worst-case* attitude. This is, at the same time, *too much* and *too little* for most systems (soft real-time, best effort, mixed criticality). It is too much because if the worst-case is rare we can live with it, as we do throughout our daily life where major catastrophes are never fully excluded. This is too little because we really want to know what will *typically* happen, not only what is possible in principle. The solution in the context of timing performance is not new: replace duration bounds which are without measure, that is, non-deterministic in the CS sense without probabilities, with probability distributions. This corresponds to the difference between Minkowski sum and convolution as shown in Fig. 5. In the set-theoretic tradition, when two tasks, both with an uncertain duration in $[a, b]$ each, are executed sequentially, the total duration can be anywhere in $[2a, 2b]$. Probabilistically, assuming a uniform distribution of the durations over $[a, b]$, a duration of $a + b$ is more likely than $2a$ or $2b$.

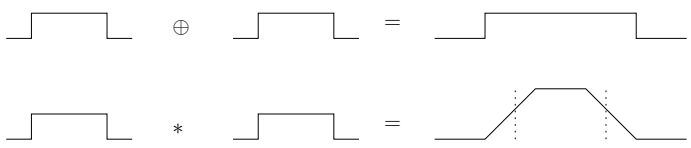


Fig. 5. From set-theoretic to probabilistic non-determinism. When two processes of duration $[a, b]$ execute one after the other, the total duration can be anywhere in their Minkowski sum $[2a, 2b]$. When the durations are assumed to be uniformly distributed, the total duration is distributed according to the convolution which still non-zero in $[2a, 2b]$ but its density is larger around the center and smaller in the tails.

The late discovery of Monte-Carlo simulation. But what can you really do with such duration-probabilistic automata? Probabilizing the timing uncertainty does not alleviate the scalability problem - on the contrary, computing probabilities over sets is typically much harder than computing the sets themselves and this is what makes probabilistic verification so difficult and essentially theoretical. One direction to think about which has not been explored to the best of my knowledge is to employ *fat first search*, exploring only reachable sets of high probability. The other solution is simply to run Monte-Carlo simulations, sample the uncertainty space and collect statistics. Then we can call it *statistical model checking* to hide the fact that after 20 years we resort finally to what practitioners have always been doing. Kurt Vonnegut had an amazing observation on these matters in *Cat's Cradle*:

> *"Beware of the man who works hard to learn something, learns it, and finds himself no wiser than before... He is full of murderous resentment of people who are ignorant without having come by their ignorance the hard way."*

If we replace exhaustive verification by Monte Carlo simulation what was the worth of the exhaustive formal verification episode? One answer is that there are still systems which are critical and require exhaustive coverage. A second answer is that every domain can always benefit from a fresh look by researchers from a different culture.

The other answer is that abstract and semantically-correct modeling, if not abused, does have advantages and can help system builders that do not possess these capabilities (rather than impress them with your knowledge of Greek letters). System builders use concrete formalisms such as C and Verilog to build their systems and this coding is unavoidable if you want the system to be constructed. Abstract models such as those used in verification or performance analysis are, first of all, considered by them as an extra burden. By the way, I cannot blame them: I don't want anyone to tell me how to hack my LaTeX code or use UML to structure my research. Secondly, many of them may have difficulties in building *abstract* models that do not correspond to something concretely executable. Consequently they use their designs as models for simulation: the software or the hardware *models itself*. When both of those already exist, this is the most efficient way to evaluate the performance (and check functional correctness). But in stages of design-space exploration when the hardware architecture or configuration is not realized, the software is run on a hardware simulator at some granularity, for example a cycle accurate simulator, and this is extremely slow. To explore different deployments it is much more efficient to use a discrete event simulator, where computations and data transfers are modeled as *timed processes* that take some time and immobilize some resource during that time. Of course you need to fill in the numbers (profiling, estimation, past experience) but recall that you need *not* be precise and deterministic.

Following these principles, the *Design-Space Explorer* prototype tool has been developed by J.-F. Kempf with help from M. Bozga and O. Lebeltel [21]. It has four components: 1) *Application description*: task-graphs annotated with execution times and data transfer rates; 2) *Input generators*: models of task arrivals (periodic, jitter, delayed periodic, bounded variability); 3) *Architecture description*: processors and their speeds, memories, busses and 4) *Deployment*: mapping and scheduling policies. From these descriptions timed automaton models are built which represent all the possible behaviors under all timing uncertainties. Then the system is analyzed using formal verification (when feasible and useful) and mostly via statistical simulation. It has been applied to compare different deployment policies for a video algorithm on a simplified model of an experimental multi-core platform developed by ST Microelectronics. Time will tell whether such modeling and analysis techniques will find their way to the design flow of dedicated multi-cores architectures and their software.

To wrap up, let me repeat once more that I consider timed automata to be one of the best inventions since the cut-and-paste. They also served as a launch pad for the study of hybrid systems. Despite the fact that they are n-tuples they can be useful, not only for the paper industry or the tool-paper industry, but to real applications. This requires more blood, sweat and tears, less theorem hunting and less bibliometry-guided research.

Acknowledgment: I would like to thank Eugene Asarin, Marius Bozga, Eric Fanchon Thomas Ferrère, Charles Rockland and P.S. Thiagarajan for commenting on various versions of this manuscript. This work was partially supported by the French project EQINOCS (ANR-11-BS02-004).

References

1. Y. Abdeddaïm, E. Asarin, and O. Maler. Scheduling with timed automata. *Theoretical Computer Science*, 354(2):272–300, 2006.

2. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
3. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.
5. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proc. IFAC Symposium on System Structure and Control*, pages 469–474, 1998.
6. F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and linearity*. Wiley New York, 1992.
7. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.
8. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12, 2006.
9. R. Ben Salah. *On Timing Analysis of Large Systems*. PhD thesis, INP Grenoble, 2007.
10. R. Ben-Salah, M. Bozga, and O. Maler. Compositional timing analysis. In *EMSOFT*, 2009.
11. B. Berthomieu and M. Menasche. An enumerative approach for analyzing time petri nets. In *Proceedings IFIP*, 1983.
12. M. Bozga, S. Graf, and L. Mounier. IF-2.0: A validation environment for component-based real-time systems. In *CAV*, pages 343–348, 2002.
13. P. Caspi. Réflexions sur la recherche appliquée. Unpublished manuscript available at http://perso.numericable.fr/bgrardca/TEXTES/recherche.pdf, 2004.
14. C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2008.
15. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems*, pages 208–219, 1995.
16. A. de Saint-Exupéry. *Le petit prince*. Gallimard, 1943.
17. D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.
18. T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
19. T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.
20. T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Real-Time: Theory in Practice*, pages 226–251. Springer, 1992.
21. J.-F. Kempf. *On Computer-Aided Design-Space Exploration for Multi-Cores*. PhD thesis, University of Grenoble, October 2012.
22. O. Maler. Amir Pnueli and the dawn of hybrid systems. In *HSCC*, pages 293–295, 2010.
23. O. Maler. On under-determined dynamical systems. In *EMSOFT*, pages 89–96, 2011.
24. O. Maler, K. Larsen, and B. Krogh. On zone-based analysis of duration probabilistic automata. In *INFINITY*, pages 33–46, 2010.
25. O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In *CHARME*, pages 189–205, 1995.
26. M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.
27. J. Sifakis and S. Yovine. Compositional specification of timed systems. In *STACS*, pages 347–359, 1996.
28. S. Tripakis and C. Courcoubetis. Extending Promela and Spin for real time.
29. S. Yovine. *Methods and tools for the symbolic verification of real-time systems*. PhD thesis, INP, Grenoble, 1993. (in French).
30. S. Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.