Some Thoughts on Runtime Verification

Oded Maler

VERIMAG
CNRS and the University of Grenoble Alpes (UGA)
Bat. IMAG, 700 av. Centrale
38041 St Martin d'Heres, France
Oded.Maler@imag.fr

Abstract. Some reflections on verification and runtime verification in general and of cyber-physical systems in particular.

1 Introduction

I was probably invited to present in this forum because of my work on checking simulation traces of cyber-physical (hybrid, mixed-signal) systems against specifications expressed using formalisms such as signal temporal logic [25] and timed regular expressions [2]. I will use the opportunity to discuss, in addition, some other issues that come to my mind in the context of runtime verification. I start in Section 2 with a reflection on the nature of words and then discuss some potential meanings of *runtime* verification as distinct from just *verification*. Section 3 describes one interpretation, the activity of monitoring simulation traces against formally defined properties. Section 4 discusses runtime verification interpreted as verification of something closer to the implementation. Section 5 is devoted to monitoring real working systems during their execution, an activity that differs in many aspects from monitoring during design and development time. Finally, Section 6 speaks briefly about robust satisfaction and the relation between properties and other quantitative performance measures traditionally applied to signals. The topics of Sections 4 and 5 are outside my domain of expertise so what I write there is based on common sense and speculation. I am sure many of these issues have been studied by researchers in this community and I apologize in advance for not having the time and resources to make a comprehensive study of relevant work before writing this document.

2 Words Speak Louder than They Should

Robert Anton Wilson, an overly-illuminated writer [32] and thinker [39], objected rather strongly to the usage of the word *is* based on some very reasonable grounds. Words are just tools, they do not have intrinsic absolute meaning and it is silly (but common) to argue about the *true* meaning of a word. The meaning depends on context and background and can differ from one occasion to the other, from one speaker or community to another and in general it shifts with time. One important aspect in studying word meanings is to consider the background against which they came to being, the specific additional distinctions and refinements of existing concepts they came to express.

As a concrete example consider the term *reactive systems* coined by Harel and Pnueli in a classical paper [14], published in 1985 some time before the emergence of CAV-style academic verification. Reactive systems are defined there as systems that maintain an *ongoing interaction* with their *external environment*. This should be understood against the background of classical computability (and complexity) theory, dealing with non-reactive (transformational) programs that map static inputs to static outputs without being in time, without interacting with the external world during computation (see some old ramblings of mine in [18]). The concept was useful in separating protocol verification (at least for some time) from other approaches to program verification.

In contrast, the term "reactive" does not have much meaning in control theory because all control systems are supposed to be reactive by definition.[1] The same holds, of course, for living systems, and those who preach reactive systems to biologists are, in fact, preaching for the introduction of discrete states and transitions into a modeling domain often dominated by continuous applied mathematics. In other contexts such as cognitive psychology, the word reactive might indicate a simple behavioral stimulus/response model, mathematically much narrower than the transducer model underlying the reactive systems of [14].

Coming to think of the possible meanings of *runtime verification* one has to think about what is particularly added by the *runtime* qualifier to the general background of the meaning of *verification*, which by itself is rather pluralistic. Verification can mean one thing to a practitioner (and this may depend on the type of software, hardware or physware development being practiced) and another thing to theoreticians of various degrees of practical inspiration, aspiration and pretention. I once received a book called *The verification Cookbook* from an EDA company and I did not find there anything remotely similar to concepts studied in most of CAV papers. Thus said, let me try to lay down the implicit semantics of verification I have been carrying in my head over the years. Needless to say after all this introduction, no claim is made for this semantics to be more truthful than any other.

So my version of verification, the algorithmic genre following [29, 3], goes like this [19]. You have a system which is open (reactive), and each of its dynamic inputs may induce a different behavior. Behaviors are viewed as trajectories (runs, paths) in the state-space, which used traditionally to be that of a large state-exploded automaton. You want somehow to ensure correctness of the trajectories induced by all admissible inputs. Correctness of a run typically means the occurrence or non-occurrence of certain temporal patterns, expressible in a declarative language (temporal logic, regular expressions) or hacked manually into property observers composed with the system.

[1] Speaking about control, "reachability" (and to some extent "controllability") used not long ago to denote some very precise technical term in the Kalmanistic theory of linear systems before some barbarians came and kidnapped its meaning. As a punishment we have sometime to hear colleagues from others disciplines abuse theoretical computer science sacred terms such as *decidability* or *models of computation*.

Rather than enumerating all inputs up to a certain length and simulating in a black box fashion, formal verification does two things. First, by having access to the automaton model of the system, the verification algorithm can follow paths in the transition graph rather than trying to cover them by a blind selection of inputs. This yields an important complexity improvement [19]. Then, some systems are sufficiently small so that modern computers can explore all their paths within a reasonable time. Otherwise, an attempt is made to reason about all the behaviors in a more holistic manner. One direction is to try to prove something about all behaviors in a deductive/analytical way, a topic I will not touch here. Alternatively, one can do a kind of set-based breadth-first simulation, which came to be known as *symbolic model checking* [26], where symbolic means that logical formulae are used to describe the set of reachable states at successive time points.

Remark: In fact, the term *model-checking* is another example of a word shared by different meanings. Today, people from outside verification to whom the concept is presented, biologists for instance, probably take it to mean checking whether your model of some physical phenomenon makes sense, for example, whether it admits some counter-intuitive behaviors not supported by experiments, whether it is robust under parameter variations and so forth. So *model* is understood as the common concept of a *mathematical* model of something in the *real world*. This is *not* the original meaning of *model* as used in model checking, which is a purely technical logical concept stating that a given mathematical structure is a model of a system of logical axioms - there is a whole branch of logic called *model theory*, which studies the relation between these two classes of mathematical objects. Model checking was initially used in verification to contrast it with deductive (proof theoretic) methods of reasoning [13]. The story for LTL (linear-time temporal logic) goes like this: a given sequence w is a model of a temporal logic formula φ if w satisfies φ. Thus verification by model checking means to check whether all possible runs are models of φ. For branching-time logics like CTL, what is checked is whether the whole transition system (Kripke structure in modal logic parlance), viewed as a generator of a branching structure, is a model of the formula. ▟

Other implicit assumptions in my story are the following.

1. The verification process takes place mostly during the *design* and *development* phase before we unleash the system to go wild in the streets and do what it is supposed to do;
2. In many cases, verification is done on a *model* of the system which is a kind of an automaton which abstracts away from data (rather than control) variables as well as some particular implementation details, including the programming language and execution platform. The more abstract this mathematical model is, that is, closer to an automaton/graph, the easier it is to reason about its behaviors in a global manner. Nevertheless, some syntactics is required to express the automaton for the purpose of verification and some connection with the program that realizes it should be eventually made;
3. The properties against which behaviors are evaluated have been traditionally *qualitative*, providing a yes/no answer concerning property satisfaction.

In the following I will contemplate on various interpretations of runtime verification by perturbing some of the abovementioned implicit assumptions. I will also discuss the adaptation of runtime verification techniques (and verification in general) to cyber-physical systems. In particular, I will touch upon the following topics:

1. Runtime verification viewed as good old simulation/testing without coverage guarantees but augmented with formal specifications;
2. Runtime verification viewed as getting closer to the real implemented artifact, further away from the abstract model;
3. Runtime indicating that we leave design/reflection time and get involved in the detection of patterns in *real* time while the system is up and running;
4. The quantitative semantics of temporal properties and the fusion of properties and assertions with other performance measures used by engineers to evaluate signals and systems.

3 Runtime Verification as Simulation plus Formal Specification

This used to be my favorite interpretation that we exported successfully to the continuous and hybrid domain [22]. Verification is a glorious activity but it does not scale beyond certain system complexity and people will always resort to simulation, with or without renaming it as statistical model checking. We also separate concerns and say that coverage and exhaustiveness is someone else's responsibility. So what remains is plain simulation with the additional twist that the simulation trace is checked against formal properties specified in some rigorous and unambiguous formalism. It can still be debated whether some engineers' reluctance to use such clean declarative languages is a bug or a feature of their way of thinking. It is related, I think, to the issue of whether you want to use the same language for implementation and specification, or rather have two distinct languages. Maybe it is easier for a developer to build a property checker as a Simulink block or a piece of C code than to learn yet another language.

According to the automata-theoretic approach to verification [38], exhaustive verification corresponds to an *inclusion* problem between two formal languages:[2] the set of behaviors produced by the system and the set of behaviors defined by the property. For checking a single behavior, the problem simplifies into a *membership* problem: does a given behavior belong to the language defined by the property? Unlike verification, this activity, that we call *monitoring* from now on, does not require a respectable mathematical model and can work with *any* black box that generates simulation traces. In fact, monitoring is agnostic about the origin of those traces, which could be as well recordings of a real system. The complexity of the system, which is a critical limiting factor in exhaustive verification, influences only the simulation time and the number of simulations needed to properly cover its behaviors, but this is, as we said, not our problem.

In the context of digital hardware, monitoring is called *dynamic* verification against *assertions* while the term *static* or *formal* verification is used for model checking. Mo-

[2] The term *formal language* provides yet another opportunity for terminological confusion. In theoretical computer science a formal language is nothing but a set of sequences, something very semantic in our context.

tivated initially by analog and mixed-signal circuits, we extended this idea[3] to continuous and hybrid systems [22, 25, 27] by introducing *signal temporal logic* (STL), which adds numerical predicates over real-valued variables on top of the dense time[4] *metric temporal logic* (MTL) [17]. We provided a simple efficient algorithm for checking satisfaction/membership for the future fragment of STL by backward interval marking. This procedure can, in principle, liberate users from the tedious task of classifying simulation traces manually by visual inspection or by writing programs for that purpose.

4 Runtime as More Real

Another interpretation of runtime is literally, *while a program is running*. This means that in contrast with the abstract automaton model, we deal here with something closer to the implementation: either we have generated real code from the abstract model or there was no such an abstract model to begin with. Software is a peculiar engineering artifact, closer in its very nature to its abstract model more than any physical system can be: compare the gap between an engine model and a real *physical* engine with the tiny gap between a model of a controller and its software implementation. For this reason, software developers may tend to skip what they perceive as a redundant modeling stage.

Cyber-physical systems admit heterogeneous components including the external environment which is modeled but not implemented, and the designed artifact itself which includes physical components, a hardware platform and software. In the development of such systems there is a multi-dimensional spectrum between abstract models and real systems, both in the implemented and unimplemented parts. This is attested by the existence of several kinds of testing, each using a different manifestations of the controller, the external environment and their interconnection. For example, hardware-in-the-loop simulation indicates that the real implemented controller, running on its execution platform is being tested. Model-in-the-loop testing, to take another example, means that the input to the implemented controller comes from a simulator, in contrast with more realistic settings where these inputs come from sensors or, at least, through real physical wires.

To perform verification while the program is running, the program should be properly instrumented to export the evolving values of the variables appearing in the property, thus producing traces that can be checked by your favorite property checker. Since we are talking about a real imperative program, not an interpreter of an automaton structure, only single runs (rather than set-based runs) can be naturally produced. This activity can still take place during the development phase (design, integration tests) but as will be discussed next, it can be applied to a working system.

[3] I am indebted to a discussion with Yaron Wolfsthal before starting this project, in which he explained to me the workings of the FOCS property checker developed at IBM for discrete/digital systems.

[4] The advantage of dense time as used in MTL or in timed automata is in not committing to a fixed time step such as the clock tick in digital circuits. Otherwise, the major advantage of timed logics and automata is *not* in density but in the ability to reason about time arithmetically rather than by counting ticks. More opinions on timed systems can be found in [21].

5 Monitoring During the System's Lifetime

The most radical departure from classical verification is obtained by interpreting runtime as meaning that we monitor real systems during their normal (and abnormal) execution. Many such systems come to mind at different scales of space and time: nuclear reactors, highway and network traffic, air conditioning systems, industrial plants, medical devices, corporate information systems and anything that generates signals and time series.

A monitoring process that is simultaneous with the ongoing behavior of the system suggests new opportunities such as detecting important events and patterns, almost as soon as they happen, and reacting to them, either by alerting a human operator or by triggering some automatic action. Well, calling these opportunities "new" is appropriate only in the verification context: such monitors exist in low-tech ever since the industrial, or at least the electrical revolution. Just consider indicators in your car control panel for speed, temperature or fuel level and more modern features like alarming the driver while getting too close to other cars or activating the airbag upon detecting a collision.

This type of application deviates, as I attempt to show, from the standard story of verification and requires rethinking of what it is the thing that we want to specify (and monitor) using our favorite formalism. To understand what I probably mean let me introduce a naive straw man, a true believer in the verification myth. According to him, if φ is the (precise) system's specifications, characterizing exactly the acceptable behaviors, the most natural thing is to tell the monitor to watch for $\neg\varphi$ and cry out loud when it occurs. But on a second thought, our straw man will add, this will not happen anyway if we verified the system and showed that all its behaviors satisfy φ. Or if you want a control version of the myth, this will not happen if the controller has been designed correctly.

To understand what is wrong here, let us first see why verification of cyber-physical systems is different, hard and, in some sense, almost an oxymoron (some related observations and discussions concerning the rigorous design of *systems*, as opposed to programs, appear in [33, 34]). The verification story is based on the following three premises:

1. You have a (very) faithful model of the system under verification;
2. You have formal requirements that indeed trace the boundary between acceptable and unacceptable behaviors;
3. The system is sufficiently small so that formal verification is computationally feasible.

The range of systems for which (1) and (2) above hold is very narrow in the cyber-physical world. It is fair to say that it is restricted to some hardware and software components, analyzed for their so called *functional* properties, those that care only about their purely computational properties, not involving physical aspects and interactions such as power consumption or timing.[5] Software is very peculiar in admitting a chain

[5] No program, no matter how thoroughly verified, will produce the correct result if you hit the computer with a hammer or just unplug it from power.

of faithful semantics-preserving models, going all the way from programs in a high-level language down to gates and transistors. Nothing like this exists in the physical world where models are understood to be just useful approximations.[6]

The same holds for specifications: you can certainly write down a complete set of properties that will characterize the valid behaviors of, say, a chip realizing some hardware protocol, verify it on an exact model and expect that the real chip will indeed work continuously without problems as long as the underlying physical assumptions hold. For systems with physical functionalities, there is typically never a comprehensive list of requirements that holds globally over the whole state-space. In fact, such a global state-space (the one-world semantics of [37]) by itself is not part of the conceptual map of most engineers. For physical systems there are domain-specific intuitions about the shape of certain response curves, the values of some quantitative measures, and so on, but you never have an explicit formalized partition of all behaviors in the huge cyber-physical state-space into good and bad behaviors. Airplanes fly, nevertheless, most of the time.[7]

So we want to use some specification formalism to express observable conditions and temporal patterns that will trigger some responses:

$$\textbf{if } \textit{some pattern is observed } \textbf{then } \textit{do the right thing.} \tag{1}$$

The entity that does the right thing can be a human operator and in that case the role of monitoring is just to create an alarm and bring the situation to her attention. If the reaction is automatic, this is yet another instance of a feedback control loop with actions based on observations, more appropriate for high-level supervisory control where discrete decisions are to be taken. Without giving a precise definition of hierarchical control, think of lower-levels controlling, say, torques and velocities in car engines or robots, essentially continuous processes and quantities, while higher levels decide whether to go right or left upon detecting an obstacle or whether to cancel the trip after observing traffic jams or fuel shortage.

Remark: Is there a particular advantage in using the format of (1) compared to standard controllers? Controllers with state variables and memory can encode in their state some abstraction of the input history that will influence their reaction. This is clearly visible for discrete-event systems where automaton states represent equivalence classes of input histories. This holds true, in principle, also for continuous controllers where you can integrate over the input signal but this is a very weak form of pattern detection. In fact, property monitors for logics such as STL are equivalent to some kind of timed automata over continuous signals that can be transformed into controllers by adding actions. ▟

[6] This fact renders our early heroic CS efforts to prove decidability results on hybrid systems somewhat misguided, at least from an applicative point of view. In one of the early hybrid systems meetings I organized in Grenoble in the 90s, Paul Caspi presented a cartoon of a dialog between a control engineer, saying: *it is trivial* and a theoretical computer scientist responding: *it is undecidable!*. But the noble activity of doing math for its own sake is common in all academic engineering domains, control included.

[7] Kurt Vonnegut's quote *Tiger got to hunt, bird got to fly; Man got to sit and wonder 'why, why, why?'* can be rephrased as *Governors govern and airplanes fly; It takes a computer scientist to wonder why.*

If we want to react, the patterns that we specify need not be the negations of complete properties but, sometimes, prefixes of those. For example, if the specification is that $x(t)$ should always remain below c, we should raise a flag when x gets alarmingly close to c and try to steer the system in the opposite direction in order to *enforce* the property (see [11] for a discussion of enforcing specifications in the discrete context). Likewise, if the specification says that every request is granted within some time-bound d, a useful monitoring systems will detect customers that already wait for some $d' < d$ time while there is a chance to serve them before the deadline.[8]

Monitoring simulation traces can be done by offline procedures that wait for the end of the simulation and then may go through the trace back and forth in both directions. For monitoring real systems we should focus on online procedures that do not wait until the *end* of the trace (which is anyway a shaky concept for real reactive systems) to start the analysis. This is technically unfortunate for the future fragment of temporal logic which is by definition acausal, with satisfaction at time t typically defined based on values at time $t' > t$. This point of view is captured nicely by temporal testers [16, 28] which are acausal transducers that provide for a compositional translation of temporal logic to automata (and timed temporal logic to timed automata [23]). Past LTL, which can express only safety properties, is causal and can report violation of a property by a prefix of the behavior as soon as it happens.

The traditional use of future temporal logic in verification is based on infinite behaviors whose time domain is $[0, \infty)$. A lot of effort, for example [9], was invested in order to define a finitary semantics, appropriate for the very nature of monitoring. One may argue that unbounded liveness is not a useful notion for monitoring and we can do with bounded-response properties whose degree of acausality and non-determinism is bounded [24]. Traditionally, properties used in verification are supposed to hold from time zero and be satisfied or violated (accepted or rejected) by the whole behavior or its prefix. For runtime monitoring it might be more appropriate to use the more general *pattern matching* concept that speaks of segments of the behavior, starting and ending at arbitrary points in time. Regular expressions seem to be more appropriate for this purpose and we have recently developed offline [35] and online [36] pattern matching algorithms for timed regular expressions [2] over Boolean signals.

6 From Quality to Quantity

Properties and assertions are functions that map behaviors (sequences, signals) into $\{0, 1\}$ according to satisfaction or violation. In many contexts, especially in the cyber-physical world, we would like to have a more refined quantitative answer: not only whether the property has been satisfied or violated by the behavior, but also how robust the answer was [10, 30, 7, 6]. For example, if we have a behavior which satisfies the requirement that $x(t)$ is always smaller than c, the distance between the maximal value

[8] Are all the things that we want to monitor restricted to prefixes of behaviors that lead to a violation of the specifications? I do not have an answer at this moment and it probably depends also on whether we are in the hard (safety critical) or soft (quality of service) domain. It is also related to whether numerical quantities are involved: the car fuel indicator shows continuously the value of a real-valued variable and, in addition, emits a warning when it crosses a threshold.

of x and c will tell us the robustness of the satisfaction, how close far we were from violation. Likewise, in a behavior w where some response has missed a deadline d, the distance between d and the maximal response time occurring in w will tell us the severity of the violation and whether it can be fixed by relaxing the specification using some $d' > d$ which is still acceptable. For a given property φ and signal w, the quantitative (robustness) semantics returns a value $\rho = \rho(\varphi, w)$ having the following two important properties:

1. The robustness ρ is positive iff w satisifies φ;
2. The φ-satisfaction of any signal w', whose pointwise distance from w is smaller than ρ, is equal to that of w.

This semantics gives more information and moreover it opens new possibilities in the search for bad behaviors, also known as bug hunting or *falsification*, which is a very active domain in the verification of cyber-physical system. The idea is that the robustness value can be used by an optimization/search procedure that explores the space of system trajectories (and the input signals that induce them) trying the minimize the robustness value until a violating behavior is found, see for example [8, 1, 31, 15, 5].

Despite these advantages, the robustness semantics still suffers from the expressive limitations of traditional logic and its orientation toward extreme-case reasoning. The quantitative semantics of STL, as defined in [7, 6], is obtained from the standard qualitative semantics by replacing Boolean values such as $x < c$ by numbers like $c - x$ and then replacing $\vee$, $\wedge$ and $\neg$ by $\min$, $\max$ and $-$. Thus the robustness value is still determined by the worst value in the signal, regardless of whether the signal spent a lot of time near that value or just had a short spike, while being at much lower values most of the time.

Many other types of quantitative measures have been traditionally applied to signals. They are based on summation/averaging, noise filtering, applying frequency-domain transforms and many other functions that extract from the signal the performance measures appropriate for the application in question. In this context, one can view STL and similar formalisms as yet another family of performance measures which excels in extracting certain features of the signal such as the sequencing of threshold crossings and other events over time, including the temporal distances between them, while being weak in terms of other features. An early attempt to combine properties and quantitative measures into a single framework is reported in [4] for discrete time. A more recent one is described in [12] where pattern matching techniques are used to define segments of the signal where standard measurements (average, extremum) are to be applied. Combining properties and measures into a unified declarative language might help in further proliferation of verification ideas [20] into the real cyber-physical world.

Acknowledgment: This text benefitted from feedback given by Eugene Asarin, Jyo Deshmukh, Jim Kapinski, Dejan Nickovic, Joseph Sifakis and Dogan Ulus.

References

1. Yashwanthand Annapureddy, Che Liu, Georgios E. Fainekos, and Sriram Sankaranarayanan. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In *TACAS*, pages 254–257, 2011.
2. Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.
3. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer Berlin Heidelberg, 1981.
4. Ben d'Angelo, Sriram Sankaranarayanan, Cesar Sanchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: Runtime monitoring of synchronous systems. In *TIME*, pages 166–174, 2005.
5. Jyotirmoy Deshmukh, Xiaoqing Jin, James Kapinski, and Oded Maler. Stochastic local search for falsification of hybrid systems. In *ATVA*, pages 500–517, 2015.
6. Alexandre Donzé, Thomas Ferrere, and Oded Maler. Efficient robust monitoring for STL. In *CAV*, pages 264–279, 2013.
7. Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *FORMATS*, pages 92–106, 2010.
8. Alexndre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *CAV*, pages 167–170, 2010.
9. Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *CAV*, pages 27–39, 2003.
10. Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.
11. Ylies Falcone. You should better enforce than verify. In *RV*, pages 89–105, 2010.
12. Thomas Ferrere, Oded Maler, Dejan Ničković, and Dogan Ulus. Measuring with timed patterns. In *CAV*, pages 322–337, 2015.
13. Joseph Y Halpern and Moshe Y Vardi. Model checking vs. theorem proving: a manifesto. *Artificial intelligence and mathematical theory of computation*, 212:151–176, 1991.
14. David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, 1985.
15. Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. Mining Requirements from Closed-loop Control Models. In *HSCC*, 2013.
16. Yonit Kesten and Amir Pnueli. A compositional approach to CTL* verification. *Theoretical Computer Science*, 331(2-3):397–428, 2005.
17. Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
18. Oded Maler. Hybrid systems and real-world computations. unpublished manuscript, 1992.
19. Oded Maler. Control from computer science. *Annual Reviews in Control*, 26(2):175–187, 2002.
20. Oded Maler. Amir Pnueli and the dawn of hybrid systems. In *HSCC*, pages 293–295. ACM, 2010.
21. Oded Maler. The unmet challenge of timed systems. In *From Programs to Systems*. Springer, 2014.
22. Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT*, pages 152–166, 2004.
23. Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to timed automata. In *FORMATS*, pages 274–289, 2006.

24. Oded Maler, Dejan Nickovic, and Amir Pnueli. On synthesizing controllers from bounded-response properties. In *CAV*, pages 95–107, 2007.
25. Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of Computer Science*, pages 475–505, 2008.
26. Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.
27. Dejan Nickovic. *Checking timed and hybrid properties: Theory and applications*. PhD thesis, Université Joseph Fourier, Grenoble, France, 2008.
28. Amir Pnueli and Anna Zaks. On the merits of temporal testers. In *25 Years of Model Checking*, pages 172–195, 2008.
29. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351, 1982.
30. Aureelien Rizk, Gregory Batt, Francois Fages, and Sylvain Soliman. A general computational method for robustness analysis with applications to synthetic gene networks. *Bioinformatics*, 25(12):169–78, 2009.
31. Sriram Sankaranarayanan and Georgios E. Fainekos. Falsification of Temporal Properties of Hybrid Systems using the Cross-Entropy Method. In *HSCC*, 2012.
32. Robert Shea and Robert Anton Wilson. *The Illuminatus! Trilogy*. Dell Publishing, 1984.
33. Joseph Sifakis. Rigorous system design. *Foundations and Trends in Electronic Design Automation*, 6(4):293–362, 2012.
34. Joseph Sifakis. System design automation: Challenges and limitations. *Proceedings of the IEEE*, 103(11):2093–2103, 2015.
35. Dogan Ulus, Thomas Ferrère, Eugene Asarin, and Oded Maler. Timed pattern matching. In *FORMATS*, pages 222–236, 2014.
36. Dogan Ulus, Thomas Ferrère, Eugene Asarin, and Oded Maler. Online timed pattern matching using derivatives. In *TACAS*, pages 736–751, 2016.
37. Pravin Varaiya. A question about hierarchical systems. In *System Theory*, pages 313–324. Springer, 2000.
38. Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, 1986.
39. Robert Anton Wilson. *Quantum Psychology: How Brain Software Programs You & Your World*. New Falcon Publication, 1990.