On the Expressiveness of some Runtime Validation Techniques

Yliès Falcone[1], Jean-Claude Fernandez[2] and Laurent Mounier[2]

[1] Laboratoire d'Informatique de Grenoble
University of Grenoble, France
`Ylies.Falcone@imag.fr`
[2] Vérimag
University of Grenoble, France
{`Laurent.Mounier,Jean-Claude.Fernandez`}`@imag.fr`

Abstract

Runtime validation techniques have been proposed as artifacts to detect and/or correct unforeseen behaviours of computer systems. Their common features is to give only partial validation results, based on a restricted set of system executions produced in the real execution environment. A key issue is thus to better understand which kind of properties can (or cannot) be validated using such techniques.

We focus on three techniques known as runtime verification, property-oriented testing, and runtime enforcement. We present these approaches at an abstract level and in a unified framework, and we discuss their respective ability to deal with properties on infinite execution sequences, that are commonly encountered in many application domains.

1 Introduction

Existing engineering methodologies are not yet able to guarantee "by construction" the correctness of real-life computer systems. To overcome their limitations, a possible solution is to rely on formal-based validation methods. Ideally, these methods should be used only in the development stages of the system, prior to any of its execution. They should also provide both *sound* and *complete* verdicts, based on some exhaustive analysis of the system behavior (described either as a model, or by the code itself). However, this ideal picture sometimes needs to be revisited on two major points, in particular:

- Scalability issues may lead to sacrifice the exhaustiveness of the analysis, leading to *partial* validation techniques;
- For specific application contexts, reliable correctness analysis may be obtained only from system executions produced and observed in a *real environment* (hardware platform, physical-world interactions, third-party components, etc.).

To address these needs, partial validation techniques operating at runtime have been proposed. In this work, we focus on three main examples of such techniques, known as *runtime verification*, *property-oriented testing*, and *runtime enforcement*. These techniques share a common feature: they produce a verdict based on the observation of a single execution sequence. This observation can be performed either *on-line*, on the running system, or *off-line*, in a post-mortem fashion (e.g., from log files). In this work we consider only the first situation, and a central issue is to better understand which kind of correctness properties can still be addressed in such a limited context.

The objective of this paper is to provide some insights into this direction. We give an account on some research efforts in that direction and propose a list of future research direction that, in our opinion, need to be investigated.

The paper is organized as follows. Section 2 presents at an abstract level the systems and properties we consider. We mainly target reactive systems, and we focus on properties on *infinite execution sequences*, expressed using well-known classification schemes like Safety-Liveness [22] or Safety-Progress [19, 8].i In Sec. 3, we present these three runtime validation techniques in a unified framework, to highlight their differences and similarities. In section 3 we summarize some existing results regarding their expressiveness, i.e., their ability to (un-)validate specific classes of properties. Finally, Sec. 5 discusses some of the challenges to be addressed to further improve these results.

2 Systems and Properties

Runtime validation techniques essentially deals with *systems* and *properties*. In this section, we present the kind of properties addressed in this paper (Sec. 2.1) and the abstract notion of system we consider (Sec. 2.2).

2.1 Properties

Formalizing and classifying properties to be satisfied by computer-based systems at execution time has been an important issue in the validation community. From a general point of view, properties aim at specifying expected system behaviours. However, various representations of behaviours can be considered.

A main distinction comes from the choice between the branching-time and linear-time views of a system execution. In the former case, properties hold on (possibly infinite) *execution trees*, which underlies an explicit notion of *state* where the system behaviour can branch to various possible futures. This gives a comprehensive way to capture non-determinism and "branching ability" (i.e., *where* choices may occur between distinct behaviors). In the linear-time view, properties hold on (possibly infinite) *execution sequences*. Each system execution is treated independently, without considering where alternative behaviour could occur. Properties addressed by runtime validation techniques based on the observation of a single execution sequence fall in this category.

Linear-time properties. Considering a set of events Σ_φ, a linear-time property φ is a subset of Σ_φ^ω ($\varphi \subseteq \Sigma_\varphi^\omega$) that defines the set of expected or authorized behaviors over Σ_φ. As such, it may specify event precedence, forbidden or expected events, whether or not events should occur infinitely often, etc.

Classifying linear-time properties. As pointed out by Pnueli in [7], these properties can be organized or classified; thus yielding an organization of the set Σ_φ^ω. Since the considered validation techniques are partial, one should take care of validating properties in the various classes, ensuring that a large spectrum of properties is covered.

The Safety-Liveness classification [15] partitions properties into two classes:[1]

safety [15] properties stating that something bad does not happen (e.g., deadlock-freedom, partial correction, FIFO ordering);

[1]Technically, the universal property Σ_φ^ω is in both classes.

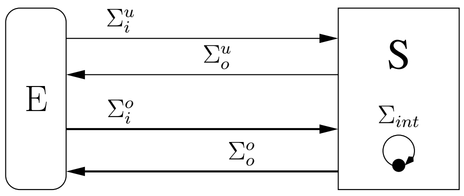

Figure 1: A reactive system interacting with its environment

liveness [1] properties stating that a good thing eventually happens (e.g., starvation-freedom, program termination).

As a consequence, when a safety property does not hold, counter-examples are *finite* sequences, whereas they are always infinite sequences for liveness properties. For more results detailing the organization of properties within this classification, we refer the reader to [1, 18, 20, 22].

The Safety-Progress classification is a hierarchy introduced by Pnueli et al. [19, 8]. It provides a finer-grain classification in a uniform way according to four views [7]: language-theoretic, logical, topological, and automata. Classes are informally defined as follows:

Safety properties are the properties for which whenever a sequence satisfies a property, *all its prefixes* satisfy this property (e.g., mutual exclusion).

Guarantee properties are the properties for which whenever a sequence satisfies a property, *there are some prefixes* (at least one) satisfying this property (e.g., total correctness, program termination).

Response properties are the properties for which whenever a sequence satisfies a property, *an infinite number of its prefixes* satisfy this property (e.g., a process willing to enter a critical section will eventually succeed, weak fairness).

Persistence properties are the properties for which whenever a sequence satisfies a property, *all but finitely many* of its prefixes satisfy this property (e.g., a process will eventually enter some nominal mode and stay inside this mode forever).

Two extra classes are defined as (finite) Boolean combinations (union and intersection) of basic classes. *Obligation* properties are combinations of safety and guarantee properties (e.g., execution of a terminating exception handler when an exception occurs). *Reactivity* properties are combinations of response and persistence properties (e.g., strong fairness). This latest is the most general class .

By expressiveness of a runtime validation technique we mean the set of properties that can be validated using this technique (and w.r.t. these classifications).

2.2 Systems

We propose a minimal common framework which is the "starting point" of the scenarios involving the addressed validation techniques.

- The techniques operate on a running system S, interacting with an external environment E through a set of actions (or events) Σ, partitioned into *input* (Σ_i), *output* (Σ_o) and *internal* (Σ_{int}) actions. Depending on the execution environment, some of these actions

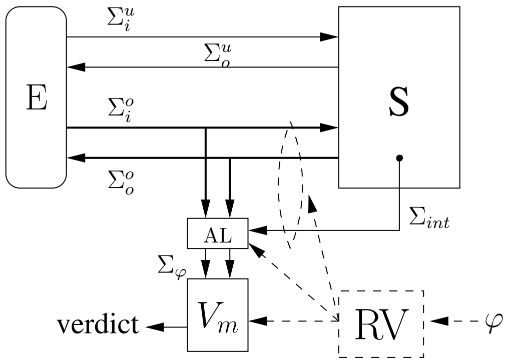

Figure 2: Runtime Verification

are considered as *observable* outside S whereas some others may be non observable. In the following we will denote by Σ_i^o (resp. Σ_i^u) the observable (resp. non observable) input actions and by Σ_o^o (resp. Σ_o^u) the observable (resp. non observable) output actions. These notations are summarized in Figure 1.

- There is no assumption about termination of the system execution: it may either terminate normally (or abort because an error occurred) after a finite number of interactions, or run forever.

A key issue is to decide on the membership of the *current* execution sequence w.r.t. a linear-time property φ by observing the observable subset of actions performed by S. Such a property defines a set of valid execution sequences over a vocabulary Σ_φ. In the general case, there might be a discrepancy between Σ and Σ_φ, for instance because the abstraction level considered to specify the property does not always correspond to the one provided by the implementation. As a result, three problems are shared by these validation techniques:

- Accessing to the input/output information exchanged by S and E, or to some internal behaviour of S, requires some *instrumentation* capabilities. These capabilities may depend both on the system reflexivity and on the instrumentation technique being used.

- In addition, an *adaptation layer* may be required to transform the observed events into elements of Σ_φ. In the following this adaptation layer is formalized by an adaptation layer AL which is usually a mapping of signature $\Sigma_i^o \cup \Sigma_o^o \cup \Sigma_{int} \rightarrow \Sigma_\varphi$.

- Deciding whether a given execution sequence σ belongs or not to φ entails to *synthesize* the corresponding procedure.

In the following we elaborate more on these three validation techniques by describing how they differently take place in the previous scenario.

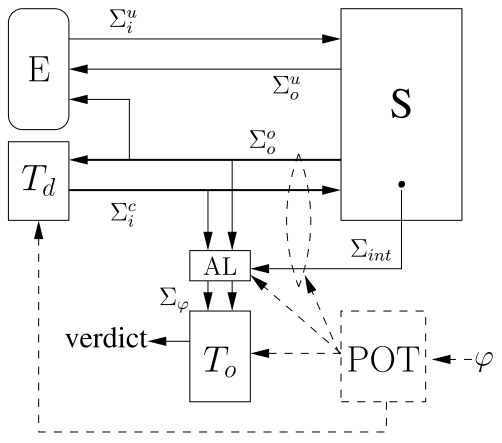

Figure 3: Property-Oriented Testing

3 Three Runtime Validation Techniques

3.1 Runtime Verification

The objective of *runtime verification* is to provide an *on-going verdict* regarding the membership of the current execution σ of S w.r.t. φ.[2] This verdict is provided by a *verification monitor* V_m from the observation of a sequence over Σ_φ (after application of the mapping AL). This architecture is depicted in Figure 2. The dashed box shows the *instrumentation and monitor synthesis* steps, performed off-line, and aiming at producing V_m, AL, and the infrastructure used to observe the interactions of S. Many solutions exist to observe the interactions and the execution of S, e.g., manual instrumentation, using Aspect-Oriented Programming technology [14], pipe systems, etc.

A general assumption is that V_m does not influence the environment E: the sequence σ is produced by S according to the current execution conditions defined by E. Therefore, this technique can be deployed on a system operating in its real execution environment.

3.2 Property-Oriented Testing

Property-oriented testing also aims to produce a verdict regarding the membership of a given set of execution sequences σ produced by S (usually called the implementation under test - IUT) w.r.t. a property φ. However, the objective here is to extend the verdict obtained from a given finite test execution[3] to a more general conclusions regarding the (non-)correctness of the whole system S itself w.r.t. φ. This is achieved by choosing test sequences σ according to some test objectives derived from φ. Thus, the execution of these test sequences is (partially) controlled by a *test driver* T_d. This test driver acts as a part of the environment E by providing to S *controllable inputs* Σ_i^c, and by receiving back the observable outputs Σ_o^o (plus maybe some

[2]First, there also exist other runtime verification frameworks dedicated for instance to the detection of concurrency errors. Second, the verdict can also be given on a recorded execution. We choose here to focus on runtime verification of user-provided specifications.

[3]or from a subset of test executions selected according to some coverage metrics.

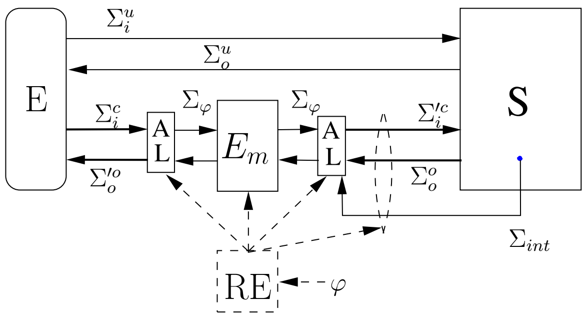

Figure 4: Runtime Enforcement

internal action Σ_{int}).[4] The decision procedure is represented by a *test oracle* T_o operating in a similar way than the monitor V_m used in runtime verification. This architecture is depicted in Figure 3. The dashed box illustrates the off-line step aiming at producing T_d, T_o, AL, and the instrumentation code (the *test harness*). This step should also include the test sequence generation, to feed the test driver.

3.3 Runtime Enforcement

The objective of runtime enforcement is rather different than the ones of the previous techniques. It consists in ensuring that the interactions between S and E are always correct w.r.t. φ (i.e., they correspond to execution sequences belonging to φ). This is achieved by the use of an *enforcement monitor* E_m, operating as a "filter" between the system S and the environment E:

- it receives the (controllable) inputs Σ_i^c produced by E and transmits some possibly new inputs $\Sigma_i'^c$ to S;
- conversely, it receives the (observable) outputs Σ_o^o produced by S and transmits some possibly new outputs $\Sigma_o'^o$ to E.

This architecture is depicted in Figure 4. The dashed box shows the *monitor synthesis* step, performed before the system execution to produce E_m, AL, and the instrumentation code. E_m is expected to operate in a *sound* and *transparent* manner. Soundness means that E_m should produce only correct execution sequences w.r.t. φ, and transparency means that it should perturb the system execution in a minimal way. Several interpretations and formalization of soundness and transparency were proposed, leading to several definitions of runtime enforcement. These interpretations differ by how the enforcement monitor is allowed to react to an incorrect input sequence (cf. [6, 9] for overviews of these definitions). One of the most studied formalization of enforcement requires that correct sequences (and longest correct prefixes of incorrect sequences) should be left unchanged (possibly w.r.t. some equivalence relation defined using φ).

[4] To simplify the notations we assume here that *controllable* and *observable* inputs are identical. This is not necessarily the case in general.

4 Some Expressiveness Results

In this section, we overview some existing results on the expressiveness of the runtime validation techniques described in the previous section. By determining the expressiveness issue for a given validation technique, we mean answering the following question:

"What properties can be used as input to the validation technique?"

Intuitively for each technique, answering this question amounts to determine how we can relate/approximate the infinite execution sequences described by the property to/by the finite sequences handled by the underlying mechanism. In the following, we address this issue assuming some simplification hypothesis w.r.t. the general framework presented in Sec. 3:

- we do not distinguish between inputs and output events;
- we do not consider any restriction in terms of observability and controlability;
- the alphabet of the underlying property Σ_φ corresponds to the set of all system events (the adaptation layer is the identity function).

4.1 Expressiveness of Runtime Verification

Viswanathan and Kim gave the first definition of monitorable properties where monitoring is purposed to detect errors on a given run [23]. The definition is motivated by the fact that an error is a deviation from a set of desired infinite sequences that should be detected on a finite execution. Moreover, the membership of an execution sequence in the set of proscribed behavior has to be decidable. Monitorable properties are thus directly defined by the authors as the set of safety properties s.t. their complement (w.r.t. the set of all possible execution sequences) is recursively enumerable. Arguably, let us remark that a verification monitor can also be used to detect "good" behaviors, i.e., the satisfaction of the desired property by the current execution. Based on this idea, Pnueli et al. gave a definition of property monitorability expressing when it is worth verifying the runtime behavior of a system [21]. The proposed definition of monitorability comes in two stages. First a property is said to be monitorable w.r.t. an execution sequence if there is a possible continuation of the program leading to a definitive evaluation of the underlying property (i.e., the current execution of the underlying program has a possible future allowing to state a verdict that cannot be further questioned).[5] This later verdict indicates either property fulfillment or violation. Second, a property is said to be monitorable when it is monitorable w.r.t. *any* possible execution of the program. No characterization (in the set of all linear-time properties) of these so-called monitorable properties was proposed. Later, Bauer et al. shown that, the set of monitorable properties is a strict superset of safety and co-safety properties [2, 3]. These classes of properties are characterized w.r.t. the Safety-Liveness classification of properties [15, 1]. The authors also gave an example of request/acknowledge property which is not monitorable.

Using the more general Safety-Progress classification, Falcone et al. proposed tighter characterizations of monitorable properties according to some of the views of the Safety-Progress classification [11, 12]. Moreover, the authors pointed out that the previous definition of Pnueli

[5]Here note that the program is more or less seen as a black box. Therefore, no assumption is made on the set of potential continuations of a given execution sequence. This set is thus assumed to be as general as possible. A simple extension of this definition would consist in e.g., restraining the set of possible futures using some "knowledge" about the behavior of the program gained from a static analysis. This would be somewhat similar to the hypothesis proposed by Beauquier et al. [4] in the context of runtime enforcement.

suffered from some practical limitations. For some properties and sequences, even when it is not possible to state a definitive verdict, it is possible to evaluate whether this sequence can be regarded as a desirable execution “if the program execution stops here”, i.e., a property is not necessarily only partitioning infinite execution sequences but also finite sequences. When the specification formalism is endowed with a finite-sequence semantics, at any moment during the system execution it is possible to decide the membership of the current execution sequence w.r.t. the desired property, provided that this membership is decidable. Based on this observation, Falcone et al. provided an alternative definition of monitorability. The alternative definition of monitorability solves the limitation pointed in the definition of Pnueli. This definition can be understood as follow. Seeing the specification formalism as a partition of execution sequences w.r.t. a truth-domain $\mathbb{D}$, this specification formalism is monitorable if it allows to associate distinct values in $\mathbb{D}$ for “good” and “bad” (finite) execution sequences. Provided that the specification formalism has a decidable finite-trace semantics, all specifications are monitorable. Moreover, for this alternative definition of monitorability instantiated with several truth-domain used so far in runtime verification, the authors characterized the set monitorable properties for ω-regular languages in the Safety-Progress classification.

4.2 Expressiveness of Property Testing

Property testability is the issue consisting in determining whether a property is *testable*. By testable we mean that the property in question can be used during a property-oriented testing activity. This property can be used for several purposes: either to directly generate test sequences, to generate test purposes so as to constrain the execution of test sequences, or to generate test oracles.

Grabowski and Namh proposed a general notion of property-testability where the set of execution sequences described by the property is compared to the set of all possible execution sequences of the IUT. The comparisons consist in determining whether an implementation relation holds between the IUT and the property. Among several implementation relations, two relations were more specifically addressed:

- the set of execution sequences of the IUT is included in the set of execution sequences described by the property, and
- the property and the IUT have at least one common execution sequence.

A property is said to be testable w.r.t. a relation if there exists a finite execution sequence allowing to determine whether the relation holds or not. The authors found that the sets of safety and guarantee properties are respectively the set of testable properties w.r.t. these relations. Intuitively, for the first one, a safety property is testable since as soon as the execution of the IUT deviates from the considered property, there is no possible future such that this execution meets again the property. Moreover, the testability of guarantee properties w.r.t. the second relation comes by duality with the testability of safety properties w.r.t. the first relation.

Later Falcone et al. [10] revisited this testability notion. In particular they identified the sets of testable properties (w.r.t. possible implementation relations) for each class of the Safety-Progress classification.

4.3 Expressiveness of Runtime Enforcement

According to how a monitor is allowed to ensure transparency, several definitions of property enforcement have been defined. Runtime enforcement monitors were thus defined as implementations of mechanisms ensuring soundness and a particular notion of transparency (as described in Sec. 3.3). Consequently, the proposed sets of enforceable properties were associated to a particular enforcement mechanisms. We shall report here the major ones[6], for more details the reader is referred to [9].

Security Automata and decidable safety properties: The pioneering work is the one of Schneider et al. proposing Security Automata as a mechanism endowed with two primitives. After reading an event, a security automaton can either produce the event in output or block (and keep forever) this event and any future event. Schneider initially announced that the set of enforceable properties with security automata is the set of safety properties. Then in [13], Hamlen et al. refined the set of enforceable properties and showed that these security automata were actually restrained by similar computational limits, as shown by Viswanathan in [23]. Hence, the set of safety properties[7] is a strict upper limit of the power of enforcement monitors defined as security automata [13].

Edit-Automata and infinite renewal properties: Ligatti et al. proposed a more "expressive" model of enforcement monitors by extending security automata with the ability of memorizing input events to output them latter. When the input sequence does not comply to the specification, it allows the monitor to "wait" for further input completing the previous bad sequence into a good one. The properties enforced by edit-automata are called *infinite renewal* properties. These properties are framed in the Safety-Liveness classification as the properties for which every infinite valid sequence has an infinite number of valid prefixes [17]. The set of renewal properties is a super set of safety properties and contains some liveness properties (but not all).

Generic enforcement monitors and response properties. One of the drawbacks of edit-automata is that, for a large number of properties, they necessitate an infinite number of control states to enforce them. Falcone et al. proposed a generic mechanism similar to edit-automata that overcome this aforementioned limitation. The so-called generic enforcement monitors (GEMs) can be instantiated with specific enforcement operations and synthesized from a declarative definition of a property. Falcone et al. showed that GEMs instantiated with a set of enforcement operations similar to insertion and suppression, can enforce the set of response properties within the Safety-Progress classification of properties. Moreover, they proved that the set of response properties is the upper-bound to any enforcement mechanism with a finite number of states (but with an unbounded memory) and that complies with the transparency constraint as described in the beginning of this section.

5 Some Future Challenges

Many research directions are still open in the context of runtime verification expressiveness. In this section, we briefly mention some of the numerous challenges related to this topic.

Observability, controlability. As explicitly mentioned in this paper, most of the approaches studying the expressiveness of runtime validation techniques do not consider any restrictive assumption regarding the observability and controlability of the system events. One

[6]without considering any memory limitation constraint of the enforcement mechanism.
[7]The unsatisfiable safety property is also not enforceable [16].

can easily imagine numerous real-life contexts where such hypothesis do not hold. For instance, most of the instrumentation techniques are limited in terms of the types of instructions they are able to automatically detect and surround with observation code. Moreover, specific inputs may not be controlled (or delayed), as assumed for test execution and runtime enforcement, without impacting too much the system execution. A related issue is the possible discrepancy between the vocabulary used to specify the property and the one provided at the instrumentation-level. Adequate abstraction/refinement operators may have to be considered to define the adaptation layer. Expressiveness results will need to be revisited in these more realistic contexts.

Distributed and parallel systems. One of the on-going challenges is to propose efficient and effective runtime validation techniques for non-sequential systems, such as multi-thread applications running on multi-core architectures, or fully distributed systems (without shared memories). The expressiveness issue raised in this paper can be transposed in such contexts. However, several questions still need to be considered, such as determining how the set of properties that can be addressed is impacted, or how to efficiently catch at runtime a consistent view of a set of concurrent execution traces with respect to a given observation criterion.

Finite vs infinite traces. As shown in this paper, a large amount of work on the expressiveness of runtime validation techniques focused on properties on infinite sequences. Indeed, the techniques addressed in this paper have their roots in exhaustive validation techniques (e.g., model-checking), traditionally based on specification formalisms dealing with infinite sequences. However, from a practical point of view, finite sequences are of first importance as well. In particular, at the implementation level, most system users are not necessarily interested by pure liveness (nor progress) properties but rather on so-called *bounded liveness* properties, expressing that "some good thing will eventually happen *within a bounded amount of time*". Such properties can be viewed as a form of safety properties, and hence can be falsified with finite counter-examples. A future research perspective would be to design an expressive specification formalisms working exclusively on finite sequences. This would entail to find what are the commonly used patterns when specifying the finite runtime behaviour of a system (as it has been addressed in the context of infinite sequences with PSL/Sugar [5]). In this respect, we believe that the set of monitorable properties would become the set of decidable properties.

Instrumentation and reflexivity. As runtime validation techniques need to instrument the underlying system, they disturb the initial behavior of the system. With the objectives of increasing the confidence in the verdict stated by those validation techniques or preserving the initial performance of the system, it is of up-most importance to disturb the system execution in a minimal way. Similarly to the expressiveness question addressed in this paper, one may wonder if it is possible to delineate a classification of enforceable properties according to how they necessitate to be intrusive w.r.t. the validated system (or conversely, how the validated system needs to be reflexive).

Acknowledgement. This paper is partly based on several previous publications, e.g., [11, 9, 10].

References

[1] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[2] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In O. Sokolsky and S. Tasiran, editors, *Proceedings of the 7th International Workshop on Runtime Verification (RV)*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138, Berlin, Heidelberg, November 2007. Springer-Verlag.

[3] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14, 2011.

[4] Danièle Beauquier, Joëlle Cohen, and Ruggero Lanotte. Security policies enforcement using finite edit automata. *Electr. Notes Theor. Comput. Sci.*, 229(3):19–35, 2009.

[5] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic sugar. In Grard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer Berlin / Heidelberg, 2001.

[6] Nataliia Bielova and Fabio Massacci. Do you really mean what you actually enforced? In *FAST'08: 5th International Workshop on Formal Aspects in Security and Trust. Revised Selected Papers*, pages 287–301, 2008.

[7] Edward Chang, Zohar Manna, and Amir Pnueli. The Safety-Progress Classification. Technical report, Stanford University, Dept. of Computer Science, 1992.

[8] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In *Automata, Languages and Programming*, pages 474–486, 1992.

[9] Yliès Falcone. You should better enforce than verify. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2010.

[10] Yliès Falcone, Jean-Claude Fernandez, Thierry Jéron, Hervé Marchand, and Laurent Mounier. More testable properties. In Alexandre Petrenko, Adenilso da Silva Simão, and José Carlos Maldonado, editors, *ICTSS*, volume 6435 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2010.

[11] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In Saddek Bensalem and Doron Peled, editors, *RV*, volume 5779 of *LNCS*, pages 40–59. Springer, 2009.

[12] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *Software Tools for Technology Transfer*, 2011.

[13] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[15] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[16] Jarred Adam Ligatti. *Policy Enforcement via Program Monitoring.* PhD thesis, Princeton University, June 2006.

[17] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, January 2009.

[18] Zohar Manna and Amir Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comput. Program.*, 4(3):257–289, 1984.

[19] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages

377–410, New York, NY, USA, 1990. ACM.

[20] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transaction Programming Languages and Systems*, 4(3):455–495, 1982.

[21] Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *FM'06: Proceedings of Formal Methods*, pages 573–586, 2006.

[22] A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In *PODC '85: Proceedings of the* 4^{th} *annual ACM symposium on Principles of distributed computing*, pages 39–48, 1985.

[23] Mahesh Viswanathan. *Foundations for the run-time analysis of software systems.* PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 2000.