A Retrospective Look at the Monitoring and Checking (MaC) Framework

Sampath Kannan[1], Moonzoo Kim[2], Insup Lee[1], Oleg Sokolsky[1], and Mahesh Viswanathan[3]

[1] University of Pennsylvania, Philadelphia, USA
[2] KAIST, Republic of Korea
[3] University of Illinois, Urbana-Champaign, USA

Abstract. The Monitoring and Checking (MaC) project gave rise to a framework for runtime monitoring with respect to formally specified properties, which later came to be known as runtime verification. The project also built a pioneering runtime verification tool, Java-MaC, that was an instantiation of the approach to check properties of Java programs. In this retrospective, we discuss decisions made in the design of the framework and summarize lessons learned in the course of the project.

1 Introduction

Motivation. The idea for the MaC project came from the realization that static verification of safety and security properties was difficult and run-time monitoring, which seemed more feasible and practical, lacked a formal framework. Program checking [5] was a relatively new and rigorous framework at that time for run-time verification of programs computing (terminating) functions computations. Our goal was to take ideas from program checking to create a formal run-time monitoring framework that would apply universally, not just to function computations, but to arbitrary programs including reactive programs that have an on-going interaction with an environment.

As a first instantiation of this goal we decided to look into run-time verification of sequential programs (e.g., C and single-threaded Java). We were presented immediately with several challenges. Because program checking was defined only for programs computing (terminating) functions, it could treat the program being checked as a black box and check only its input-output behavior. In contrast, we were interested in checking properties over program behavior during execution. Since we were monitoring and checking stateful programs, our monitors needed to keep track of the values of variables in the programs being checked.

Next, in order to check the correctness of a program, one needs to have a notion of correctness defined independently of the program. Program checking had been successfully used largely for functions whose correct behavior was mathematically defined. Examples included functions in linear algebra such as

matrix rank or matrix product, graph-theoretic functions such as graph isomorphism, and optimization problems such as linear programming. Not only did these functions have rigorous mathematical definitions of correctness, but they also admitted 'local' self-consistency conditions. For example if two graphs are not isomorphic, an isomorphic copy of one of them is not isomorphic to the other. To design a program checker, one proved the sufficiency of such local consistency checks for proving the correctness of these functions and then implemented a checker using local self-consistency checks.

Problems. In checking arbitrary programs, however, we would not have a simple exogenously-defined, mathematical notion of correctness. How then were we going to impose what correct behavior meant? For this we turned to formal methods, and specifically model checking, where such notions of correctness were defined using temporal logics such as CTL and LTL, and automata.

What makes a correctness specification in one of these formalisms truly different from a direct and step-by-step correctness specification of a program? For if the latter were the way correctness was specified, then the specification would be very specific to a particular implementation and programming language used to write the program. The key distinction between the specification and the program was the level of abstraction or detail. Correctness properties in temporal logic are generally specified in terms of permissible sequences of occurrences for certain high-level or abstract events, while the program's behavior depends on low-level details such as the values of variables and the changes to these values in the course of execution of the program.

Solutions. Regarding how we relate the detailed behavior of the program to the high-level events in the specification, which was a major design challenge, one of the important design decisions was to let the designer of the program specify these relationships, rather than seeking to automate the process of discovering them. Thus the programmer, who would be intimately familiar with the details of the program would identify the variables whose values and value changes would trigger high-level events. The programmer would also provide a logical specification of when a high-level event occurs. The MaC framework would provide the language for expressing these logical connections.

We had to decide how events would be expressed in terms of values of variables. We realized that, for example, an event could be triggered at the *instant* at which some variable changed its value, but only if it happened during the *duration* that another variable had a particular value. Thus, we needed primitive variables both for describing instantaneous changes and durations. The specific logic we used to combine these variables to describe events will be described in the sequel.

There were many other design decisions, some in setting up the conceptual framework, and some that arose when we implemented a system based on this framework. Again, we describe some of these choices in the sequel.

In the rest of this paper we describe the timeline of the MaC project, some of the detailed objectives of the project, and design decisions we made, and the impact the project has had.

Timeline of the MaC project. The Monitoring and Checking (MaC) project started as part of ONR MURI funded during 1997-2002. Goals of the MURI project were to make advances in software verification with specific applications to cyber-security. One of the initial ideas was that the well-known program-checking hypothesis [5], namely that it is often more efficient to check the correctness of a result than actually generating the result, can be applied to program correctness verification. First publications of describing the framework architecture and design trade-offs appeared in 1998 [12] and 1999 [15, 17] and the initial version of the tool, Java-MaC, implementing the MaC framework for monitoring of Java programs, has been presented at the first workshop on Runtime Verification in 2001 [13]. Since then, several extensions to the monitoring language and tools have been incorporated, while keeping the architecture intact. As the most significant extensions, we mention the steering capability [14], parametric monitoring [24], and support for monitoring of timing and probabilistic properties [21]. The Java-MaC tool has been applied to a variety of case studies, including an artificial physics application [9], network protocol validation [4], and a control system application that provided a simplex architecture-like effect using steering [14]. A variant of the tool to generate monitors in C has been applied to monitor a robotic control system [25].

Objectives of the MaC project. The MaC project has several distinct objectives from its inception:

- Understand requirements for formal specification to represent monitorable properties and choose or develop a suitable language;
- Understand requirements for a tool infrastructure for monitoring and checking of software systems with respect to formal properties and develop and architecture to help satisfy these requirements; and
- Develop a prototype tool for software monitoring and checking.

All of these objectives were achieved in the course of the project. In the rest of the paper we will discuss design decisions that were made in the process.

Overview of the MaC architecture. A visual representation of the architecture for the MaC framework is shown in Figure 1. The architecture has two tiers. The top tier represents design-time activity. The user specifies properties using the MaC languages. There is a clear separation between primitive events and conditions, defined directly in terms of observations on the system, and derived events and conditions, defined hierarchically in terms of simpler objects. This separation is also maintained at run time in the lower tier of the architecture diagram, where a component called monitor or event recognizer observes the execution and detects occurrence of primitive events and changes in the values of primitive conditions. The checker then operates on the stream of primitive events and

determines whether the property is satisfied. The definitions of primitive events serve an additional purpose: they capture, which observations on the system are important for monitoring. This information is then used to instrument the system to provide required observations. Finally, the checker can raise alarms to notify system operators or provide feedback to the system through additional instrumentation or via an existing recovery interface.

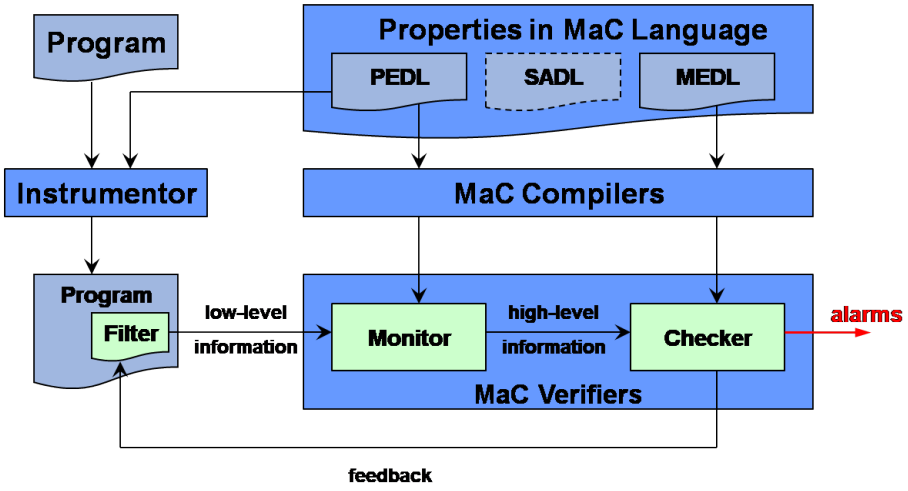

Fig. 1. Architecture of the MaC framework

2 MaC design highlights

In this sections, we take a closer look at components of the MaC framework and key design considerations for them. We consider property specification in the design-time layer of the framework, architecture of the run-time layer, and feedback capabilities.

2.1 Specification languages and their semantics

Two-tiered specification. As mentioned above, the MaC framework includes two specification languages: Primitive Event Definition Language (PEDL) and Meta-Event Definition Language (MEDL). This approach allows for separation of concerns: behavior specification is expressed in MEDL, in terms of abstract notions such as events and conditions. Separately, primitive events and conditions are defined in PEDL in terms of program entities such as function calls and variable assignments. The PEDL language is by necessity specific to the system being monitored, since event definitions need to refer to system entities. For example,

in Java-MaC, an instantiation of the MaC framework for Java programs, PEDL expressions operate on method calls, updates to fields of an object, or local variables within a method. Objects are referenced using the "dot" notation familiar to Java programmers. By contrast, MEDL is intended to be system-independent.

A distinctive feature of MEDL is that it allows users to intermix two specification styles: a logical specification based on a past-time temporal logic and operational specification based on guarded commands over explicitly defined state variables. The interplay between the two specification styles is further discussed below.

Continuous-time semantics. MEDL specifications express properties of an executions at all time instances, not just instances where observations are available. This is in contrast to most RV approaches, where semantics of a specification are given in terms of a trace, i.e., samples of an execution captured by available observations. The consequence of defining semantics in terms of a given trace is that the question of whether we check the right trace is left out of the problem. To match MEDL specifications to program execution, the set of primitive events in a MEDL specification imposes requirements on what observations need to be extracted, and a PEDL specification describes how the extraction should be performed. We can easily check that every primitive event has a definition. If the right instrumentation technology is available, the PEDL specification also becomes the basis for automatic instrumentation.

Events and conditions. The core of the MEDL language is the logic of events and conditions. Events and conditions are two distinct semantic entities in the logic. Events are instantaneous and signal changes in the state of the system. Typical examples of events are function calls and returns and assignment of values to variables. By contrast, conditions can be thought as predicates over the state of the system. Conditions evaluate to Boolean values and persist for a certain interval of time. Events and conditions as software specification devices have been around since the work of D.L. Parnas [1] and implemented in tools such as SCR$*$ [11].

Most logics defined in the literature avoid making this distinction (e.g., duration calculus [30]) or concentrate on one or the other notion. State-based logics capture system properties in terms of states, while action-based logics concentrate on state changes. It is well-known that one specification style can be transformed into the other (see, e.g., [7]). In a monitoring setting, where properties are checked over a discrete trace, in which states are comprised of observations, it is indeed tempting to treat events as predicates. Such a predicate would be true in states where the event is observed and false everywhere else. Such a view would allow us to treat events and conditions uniformly. Nonetheless, we chose to treat events and conditions as semantically different kinds in the logic for the two reasons discussed below.

While, theoretically, it is sufficient to have either state-based or logic-based approach, they result in different specification styles. We believed that different kinds of system properties are more naturally specified using different styles.

Moreover, it may be helpful to combine state-based and event-based reasoning, resulting in more compact and understandable specifications.

Second, we wanted to make claims about satisfaction of properties not just at instances when observations are available, but at all time instances. When we try to do this, we notice that conditions and events require very different reasoning. If, at a certain time point, there is no observation for an event, we conclude that the event is not occurring at that time point. By contrast, if there is no observation to evaluate the predicate of the condition, we conclude that the value of the predicate has not changed since the last time the predicate has been evaluated. If we tried to use the uniform representation of both events and conditions as predicates, as suggested above, we would not be able to properly choose the reasoning rule. To avoid this problem, we define separate semantic definitions for events and conditions.

The intuition presented above, treating conditions as abstractions of state and events as abstractions of state changes, allows us to define relationships between events and conditions. Each condition c, primitive or composite, gives rise to two events, $\texttt{start}(c)$ and $\texttt{end}(c)$. These events occur when c changes its value: $\texttt{start}(c)$ occurs at the instance when the predicate defining c becomes true and $\texttt{end}(c)$ occurs when the predicate defining c becomes false. Conversely, given two distinct event definitions e_1 and e_2, we can define the condition $[e_1, e_2)$, which is true at the current time if there has been an occurrence of e_1 in the past, but no occurrence of e_2 between that occurrence and the current moment. We refer to $[e_1, e_2)$ as the *interval* operator and note that it is similar to the *since* operator in past-time LTL.

The interval operator $[e_1, e_2)$ is the only temporal operator of the core logic of MEDL. From the discussion above, it is clear that it is a past-time temporal operator, with semantics given in terms of the prefix of the execution trace seen so far. This design decision was motivated by two considerations. First, we focused on detecting violations of safety properties and it is well known that if a safety property is violated, a violation is always exhibited by a finite prefix of an execution, so a past-time logic was deemed an appropriate specification approach. Second, a past-time approach allows us to avoid reasoning about future extensions of the current prefix and dealing with uncertainty about the future. In turn, this lack of uncertainty leads to more efficient checking algorithms. Processing a single observation takes time linear in the size of the formula and is independent of the length of the observed trace, which matches the complexity of checking past-time LTL [10]. The amount of space needed to represent the state of the monitor is also linear in the size of the formula and can be determined statically while generating the monitor.

Three-valued logic. Both specification languages of MaC framework are based on a three-valued logic to express *undefined* states of a target program in a compact manner. For example, a member variable v_j of an object o_i may not be visible until o_i is instantiated. In such situation, an expression e_k of behavioral specification like $\texttt{o}_i\texttt{.v}_j\texttt{==10}$ is undefined. This expression becomes defined only after $\texttt{o}_i$ is instantiated. Similarly, this expression becomes undefined again if $\texttt{o}_i$

is destructed. Thus, an expression of behavioral specification may change its definedness during the execution of a target program and three valued logic of PEDL/MEDL can conveniently describe such changes.

Monitor state and guarded commands with auxiliary variables. In addition to the logic of events and conditions, MEDL specifications can include guarded commands. Commands are sequences of expressions that update state variables of the monitor. We refer to these state variables as *auxiliary variables*, since they extend the state of the monitored system. Commands are triggered by occurrences of events defined in PEDL or MEDL. In turn, auxiliary variables can be used in predicates that define MEDL conditions and, ultimately, define new events. This creates a potential for infinite loops in monitor execution. MEDL semantics have been augmented to detect potential loops and reject such specifications as invalid.

2.2 Tool architecture

Instrumentation vs. virtual machine. In order to support the continuous-time semantics defined above, instrumentation has to guarantee that no changes to monitored variables are missed. As a different method of extracting runtime information, we can utilize a monitoring and checking layer on top of a virtual machine such as JVM or LLVM virtual machine through debugging interfaces (e.g., The Java Virtual Machine Tools Interface (JVM TI)). Although a virtual machine-based approach can extract richer runtime information than the one extracted through target program instrumentation, it might be slower than the target program instrumentation. Also, at the time of developing Java-MaC (i.e., 1998-2000), JVM did not have "good" debugging interface, and thus, we determined that it would have required significantly more amount of effort to develop Java-MaC as a virtual machine layer than to develop Java-MaC as a framework to instrument a target program.

Bytecode-level vs. source-level instrumentation. To extract runtime information of a target program, a monitoring framework can instrument a target program either in a bytecode (i.e., executable binary) level or a sourcecode level. We decided to instrument a target program in a bytecode-level for the following reasons:

- *high applicability* (i.e., can be applied to almost all target programs)
- *fast setup* for runtime verification (i.e., no source code compilation required).
- *on-the-fly applicability to mobile applications* (e.g., Android applications) which are downloaded from app stores (e.g., Google playstore).

The weakness of bytecode level instrumentation is that it is difficult to directly obtain high-level runtime information from a target program execution. However, we believe that PEDL and MEDL scripts can enable reconstruction of high-level behavior of target program executions based on the low-level monitored data. In contrast, source-level instrumentation can be very complicated

depending on the complexity of target source code, since the instrumentation should handle all possible complex statements of a target program.

Asynchronous vs. synchronous monitoring. Although MaC architecture can be applied to synchronous as well as asynchronous monitoring, our Java-MaC tool was designed to operate asynchronous monitors. The motivation for this design decision was to reduce timing overhead, i.e., disruption to the timing behavior of the system: instead of stopping the system while an observation is processed by the monitor, we send the observation to a stand-alone monitor, allowing the system to move along. Although the instrumentation to extract observation still needs to run as part of the system, checking of the property is performed elsewhere.

Checking of timing properties. When dealing with properties that specify quantitative timing behavior, the monitor needs to keep track of the progress of time. If an event e_2 should occur within a certain interval of time after an occurrence of e_1, the monitor needs to detect that the interval has expired. With the focus on asynchronous monitoring, timing properties present additional challenges in the MaC architecture, since the monitor clock may be different from the system clock. One can rely on timestamps of observations received from the system. Assuming in-order event delivery, once an observation with a sufficiently large timestamp is received, the monitor can conclude that e_2 did not occur in time. There may be a delay in detecting the violation, which may or may not be acceptable. However, if there is a possibility that observations will stop arriving if e_2 misses its deadline, then the violation will never be detected. In that case, the monitor would be required to track progress of time using its own clock, which requires additional assumptions about clock synchronization, delays in transmitting observations, etc. Extensions to the MEDL language and ways to provide guarantees of timely detection have been studied in [22].

2.3 Response

When a violation of a property is detected, it is not sufficient to just raise an alarm. Human operators may not be able to respond to an alarm fast enough, may not have sufficient situational awareness to choose an appropriate action to take, or may not have the right level of access to the running system. The MaC architecture allows the monitor to decide on the action and provides an interface to apply the action though the same instrumentation technology used to extract observations. We referred to this capability as *steering*. In response to an event raised by the monitor, a *steering action* can be performed to change the state of the running system or to invoke a recovery routine that may be provided by the system. A general theory of steering that would allow us to reason about the effects of monitor-triggered actions is not available. However, several case studies showed the utility of steering in situations where a high-level model of the system behavior is available. In particular, in [14], we developed a monitor-based implementation of Simplex architecture [23] and demonstrated

its utility in a control system. In [9], a simulation-based study illustrated efficacy of steering in a distributed robotic application based on artificial physics.

3 Lessons learned

After more than two decades of working on runtime verification problems, we can look back at the MaC framework and assess its vision and design through the prism of accumulated experience. We see two kinds of lessons that can be learned, as discussed in detail below. First, we can look at the impact of design decisions we have made and compare them with alternative decisions and possible extensions we did not pursue. Second, we can revisit our vision for how run-time verification would be applied and contrast it with emerging practical applications.

3.1 Reflections on MaC design decisions

Probably the most significant contribution of the MaC project was to perform an exploration of design choices in runtime verification, before settling on a particular set of decisions. We revisit some of these decisions below and briefly compare them with alternative approaches taken by the research community.

The separation of MEDL and PEDL. Separation of event definition from the rest of the monitoring specification proved very useful and we believe it is one of the most important insights to come out of the MaC project. It allows to quickly adapt to changes both in properties to be checked and in system implementations. On the one hand, if a change to the property does not require any new primitive events, there is no impact on system instrumentation. However, if we are unable to represent the changed property with existing primitive events, we know that a new primitive event needs to be introduced, which in turn tells us exactly what new instrumentation is needed. On the other hand, if a system implementation is changed, we just need to update the definition of primitive events and the rest of the monitoring setup is not affected. In this way, primitive event definitions serve as a layer of abstraction, isolating checkers from the system itself to the extent possible. In the case of software monitoring, primitive event definitions are relatively straightforward and are defined in terms of function calls and returns and state updates. However, in many situations where direct observation is more difficult, in particular in cyber-physical systems where continuous environments need to be monitored. Here, event detectors need to deal with noisy observations, for example, using statistical techniques. In such cases, a clear separation between properties, checked in a conventional way using logics, and statistics-based detection of primitive events is even more useful. Preliminary investigation of such a setting has been explored in [20].

At the same time, it gradually became clear that separation between primitive events and the rest of the event and conditions used within the monitor may be rather arbitrary. In fact, a complex system may benefit from multiple

levels of abstraction, where events and conditions on one level are defined in terms of events and conditions at levels below. This insight became one of the foundations in our follow-up work on modular runtime verification systems [29].

MEDL vs. LTL, past time vs. future time. Many people prefer to work with familiar temporal logics like LTL. Since LTL is a future-time logic that has its semantics over infinite traces, runtime verification requires additional machinery to reason about all possible extensions of the currently observed prefix. Elegant approaches have emerged after the conclusion of the MaC project, e.g., [2], which is based on three-valued semantics of LTL. In addition, such an approach allows us to easily decide when it is possible to turn the monitor off because the outcome of checking will not change in any future extension of the trace, something that is not always easy to do with past-time formulas.

Monitorability. Our approach in the MaC framework was to view runtime verification as an approach to detect *violations* of specifications. This means that *monitorable* properties would have to be safety properties, that have finite witnesses demonstrating their violation. Further, any checking framework can only detect safety properties whose set of violating prefixes is a recursive set. It turns out that the MEDL language (and its translation to automata) is as powerful as one can hope for — the MaC framework can detect violations of all safety properties whose set of violating prefixes are decidable [26]. Since this initial work on understanding the expressiveness of what can and cannot be monitored, subsequent work has identified richer notions. In this work, one views runtime verification as not just an approach to detect specification violations, but also as a means to establish that an observed execution is guaranteed to meet its specification for all future extensions of the observed prefix [2]. Such properties (i.e., those that can be affirmed) need to be such that a witnessing finite prefix of an execution guarantees their satisfaction; these are the class of *guarantee* or *co-safety* properties. The notion of monitorable properties has been further extended in [18].

Temporal logic vs. abstract commands. The mixture of temporal logic constructs and guarded commands in the monitoring language makes the approach more expressive, but complicates semantics due to the presence of potentially circular dependencies. State of the monitor is now spread between explicitly introduced state variables and values of conditions defined in the logical part of the language. Understanding the property being checked may now require more effort by the user.

Synchronous vs. asynchronous monitoring. The focus of Java-MaC on asynchronous monitoring turned out to be one of the design decisions that, in retrospect, was not completely justified. Support for synchronous monitoring turned out to be useful in many situations, in particular for security properties as well as checking timing properties in real-time. Moreover, case studies suggest that

asynchronous monitoring may not always reduce timing overhead. With asynchronous monitoring, instrumentation probes do not perform checking directly, but instead have to deliver collected observations to the monitor. When the monitor is running in a separate process or on a remote computing node, the overhead of buffering and transmitting observations often turns out to be higher than performing checks synchronously within the instrumentation probe. To the best of our knowledge, there has been no systematic exploration of the trade-off between synchronous and asynchronous deployment of monitors. Preliminary results are available in [28].

Randomization. As mentioned in the introduction, the original motivation for the work in the MaC project, was to extend ideas from program checking [5] to checking reactive program computations. In the context of program checking, randomization is often critical to obtain effective checkers. Does the same apply in the context of runtime verification of reactive programs? More recent work has tried to exploit randomization in the context of runtime verification [6, 16], including identifying the theoretical limits and expressiveness of such checkers [6].

3.2 Applications of runtime verification in safety-critical systems

Recurrent questions about runtime verification technologies concern which properties it makes sense to check at run time and why they were not verified at design time. As part of our original motivation for the MaC project, our answer to these questions was that properties come from system requirements, but they could not be formally verified at design time because state of the art in formal verification did not scale sufficiently well. For a safety-critical system this vision seems insufficient. Discovering a violation of a safety property during a mission does not improve safety, as it may be too late to react to an alarm. Therefore, more realistic approaches need to be applied to make sure that runtime verification improves safety assurance. Without trying to be exhaustive, we consider three such approaches below.

Predictive monitoring. While discovering a safety violation after it occurs may not be acceptable, discovering that a violation is imminent would be very desirable. To achieve this capability would requires us to predict likely executions in the future for a limited horizon. Such prediction may be difficult for software executions. However, for cyber-physical systems, where an accurate model of system may be available, model-based predictions are able to achieve this goal. The challenge is to keep the approach computationally feasible, due to inherent uncertainties in the model and noisy observations. A promising approach [27], based on ideas from Gaussian process regression theory, appears to be efficient enough to be applied on small robotic platforms.

Monitoring-based adaptation. Finally, an important use case is when the outcome of monitoring is used to take action aimed at helping the system recover from the problem or adapt to a new situation. These actions can take different

forms. In our early work, we showed that the well-known control-theoretic approach based on Simplex architecture [23] can be implemented in a monitored setting [14]. This case targets faults in controllers, where the checker monitors boundaries of the safety envelope of the system and triggers a switch to a safety controller, which may have worse performance but is trusted to keep the system safe. This approach relies on careful control-theoretic analysis of the system dynamics and targets a limited case when the source of the fault is assumed to be known and the action is pre-determined. In more general scenarios, several alternative approaches have been considered. One approach is to avoid diagnosing the problem, concentrating instead on ensuring that observable behavior is acceptable. This approach came to be known as runtime enforcement. Rather than altering the state of system components to allow them to return to correct operation, runtime enforcement concentrates on making sure that observable behavior is safe. Runtime enforcement actions involve delaying, suppressing, or modifying observations in other ways. A different approach is to diagnose the problem and localize the fault by collecting additional information and invoke an appropriate existing recovery procedure or applying a change directly to the internal state of a faulty component. Providing guarantees in the latter approach may be difficult and requires an accurate model of system components. A detailed survey of state of the art is given in [8].

Monitoring of assumptions. In open systems that have to operate in environments that are not sufficiently known, verification is typically performed with respect to assumptions about the environment. In this case, it is important to detect that some of the assumptions are violated at run time. We note that a violation of the assumption does not necessarily indicate an immediate problem. The system may still be able to successfully operate in the new environment. However, some of the design-time guarantees may not hold any more and system operators should pay additional attention to the situation.

In some approaches, most notably in assume-guarantee frameworks for reactive systems [3], assumptions – just like guarantees – can be naturally expressed in specification languages such as LTL or MEDL. In many other cases, assumptions take drastically different forms. For example, in control systems, assumptions are often made about the levels of noise in sensor streams. Similarly, learning-based systems rely on assumptions about training data, in particular that training data are assumed to be drawn from the same distribution as inputs encountered at run time. Detecting violations of such assumptions require statistical techniques. While there is much literature on statistical execution monitoring in process control and robotics (see, e.g., [19]), treatment of statistical monitoring tends to be much less formal than logic-based monitoring. Much work remains to be done to determine monitorability conditions for statistical monitoring and develop specification languages with formal semantics.

Acknowledgement

We would like to thank Dr. Ralph Wachter who provided and encouraged us with research funding and freedom to explore and develop the MaC framework when he was at the ONR. We also would like to thank other participants of the ONR MURI project: Andre Scedrov, John Mitchell, Ronitt Rubinfeld, Cynthia Dwork, for all the fruitful discussions. Recent extensions of our monitoring and checking approach have been funded by the DARPA Assured Autonomy program under contract FA8750-18-C-0090, and by the ONR contract N68335-19-C-0200. One of the authors of the first MaC paper [12], Hanêne Ben-Abdallah, participated in the project as a summer visitor in 1998.

References

1. Alspaugh, T.A., Faulk, S.R., Britton, K.H., Parker, R.A., Parnas, D.L., Shore, J.E.: Software requirements for the A7-E aircraft. Tech. Rep. NRL Memorandum Report 3876, Naval Research Laboratory (Aug 1992)
2. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering Methodologies **20**, 14:1–14:64 (2010)
3. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: Formal Methods for Components and Objects: 6^{th} International Symposium, FMCO 2007. pp. 200–225 (Oct 2007)
4. Bhargavan, K., Gunter, C.A., Kim, M., Lee, I., Obradovic, D., Sokolsky, O., Viswanathan, M.: Verisim: Formal analysis of network simulations. IEEE Trans. Software Eng. **28**(2), 129–145 (2002). https://doi.org/10.1109/32.988495, https://doi.org/10.1109/32.988495
5. Blum, M., Kannan, S.: Designing programs that check their work. Journal of the ACM **42**, 269–291 (Jan 1995)
6. Chadha, R., Sistla, A.P., Viswanathan, M.: On the expressiveness and complexity of randomization in finite state monitors. Journal of the ACM **56**(5), 26:1–26:44 (2009)
7. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Semantics of Systems of Concurrent Processes. pp. 407–419 (1990)
8. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Lectures on Runtime Verification, Lecture Notes in Computer Science, vol. 10457, pp. 103–134. Springer (2018)
9. Gordon, D., Spears, W., Sokolsky, O., Lee, I.: Distributed spatial control and global monitoring of mobile agents. In: Proceedings of the IEEE International Conference on Information, Intelligence, and Systems (Nov 1999)
10. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS'02). LNCS, vol. 2280, pp. 342–356 (Apr 2002)
11. Heitmeyer, C.L.: Software cost reduction. In: Encyclopedia of Software Engineering. John Wiley & Sons, Inc. (2002)
12. I.Lee, H.Ben-Abdallah, S.M.O.M.: A monitoring and checking framework for runtime correctness assurance. In: Proceedings of the Korea-U.S. Technical Conference on Strategic Technologies (Oct 1998)

13. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: Java-MaC: a run-time assurance tool for Java programs. In: Proceedings of Workshop on Runtime Verification (RV'2001). Electronic Notes in Theoretical Computer Science, vol. 55 (July 2001)
14. Kim, M., Lee, I., Sammapun, U., Shin, J., Sokolsky, O.: Monitoring, checking, and steering of real-time systems. In: 2^{nd} Workshop on Run-time Verification (Jul 2002)
15. Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., Sokolsky, O.: Formally specified monitoring of temporal properties. In: Proceedings of the European Conference on Real-Time Systems (ECRTS '99). pp. 114–121 (Jun 1999)
16. Kini, D., Viswanathan, M.: Probabilistic automata for safety LTL specifications. In: Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 118–136 (2014)
17. Lee, I., Kannan, S., Kim, M., Sokolsky, O., M.Viswanathan: Runtime assurance based on formal specifications. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (Jun 1999)
18. Peled, D., Havelund, K.: Refining the safety-liveness classification of temporal properties according to monitorability. In: Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 11200, pp. 218–234 (2018)
19. Pettersson, O.: Execution monitoring in robotics: A survey. Robotics and Autonomous Systems **53**(2), 73–88 (2005)
20. Roohi, N., Kaur, R., Weimer, J., Sokolsky, O., Lee, I.: Parameter invariant monitoring for signal temporal logic. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control. pp. 187–196 (2018)
21. Sammapun, U., Lee, I., Sokolsky, O., Regehr, J.: Statistical runtime checking of probabilistic properties. In: Proceedings of the 7^{th} Workshop on Run-time Verification. Lecture Notes in Computer Science, vol. 4839, pp. 164–175 (Mar 2007)
22. Sammapun, U.: Monitoring and checking of real-time and probabilistic properties. Ph.D. thesis, University of Pennsylvania (2007)
23. Sha, L.: Using simplicity to control complexity. IEEE Software **18**(4), 20–28 (July/August 2001)
24. Sokolsky, O., Sammapun, U., Lee, I., Kim, J.: Run-time checking of dynamic properties. In: Proceeding of the 5th International Workshop on Runtime Verification (RV'05). Edinburgh, Scotland, UK (July 2005)
25. Tan, L., Kim, J., Sokolsky, O., Lee, I.: Model-based testing and monitoring for hybrid embedded systems. In: Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration (IRI '04). pp. 487–492 (Nov 2004)
26. Viswanathan, M., Kim, M.: Foundations for the run-time monitoring of reactive systems: Fundamentals of the MaC language. In: International Conference on Theoretical Aspects of Computing (ICTAC). LNCS, vol. 3407, pp. 543–556 (Sep 2004)
27. Yel, E., Bezzo, N.: Fast run-time monitoring, replanning, and recovery for safe autonomous system operations. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS) (Nov 2019), to appear.
28. Zhang, T., Eakman, G., Lee, I., Sokolsky, O.: Overhead-aware deployment of runtime monitors. In: In this volume. (Oct 2019)
29. Zhang, T., Eakman, G., Lee, I., Sokolsky, O.: Flexible monitor deployment for runtime verification of large scale software. In: International Symposium on Leveraging Applications of Formal Methods. pp. 42–50. Springer (2018)
30. Zhou, C., Hansen, M.: Duration Calculus. Springer (2004)