Advance Access publication on 16 May 2012 doi:10.1093/comjnl/bxs042

A Performance-Oriented Monitoring System for Security Properties in Cloud Computing Applications

ANTONIO MUÑOZ*, JAVIER GONZALEZ AND ANTONIO MAÑA

GISUM Group, Department of Languages and Computer Science, University of Malaga, Malaga, Spain
**Corresponding author: amunoz@lcc.uma.es*

Security is considered one of the crucial issues for the widespread adoption of cloud computing. Despite all research done in preventive security for cloud computing, the high complexity and the interdependence of many software layers and infrastructures mean that in practice there are always chances for something going wrong. For this reason, there is a need to complement preventive security measures with reactive measures. Among these, monitoring is the most relevant approach. In this paper, we introduce a new and robust architecture for dynamic security monitoring and enforcement specially designed for cloud computing scenarios. Our solution is therefore a complete one including a three-layered architecture, a new language for expressing monitoring rules and a strategy based on the generation of a finite-state machine to improve the performance of the monitoring engine.

Keywords: cloud computing; monitoring rules; dynamic verification; security properties; event-sequence language

Received 5 December 2011; revised 25 March 2012
Handling editor: Bharat Jayaraman

1. INTRODUCTION

In a perfect world, applications and security mechanisms would be correctly designed to fulfill the user's needs thanks to flawless implementations. In real world, the design of applications and security mechanisms is sometimes based on wrong assumptions, oversimplified facts, vague, incorrect and incomplete requirements, etc., which are, in most of cases, poorly implemented. In this setting, monitoring does not only make sense but also becomes essential in order to achieve high levels of assurance.

Obviously the role of monitoring is to complement the security elements already included in a system (via hardware, OS, platform/middle-ware...) by establishing assumptions about the system security and then actually checking for such assumptions hold true at runtime [e.g. we can assume that the different Virtual Machines (VMs) running in a platform are isolated]. Additionally, monitoring can be useful for different purposes. It can help in increasing trust in infrastructures and applications, checking for compliance (e.g. an e-government process that has to be split in phases and has to produce records of the intermediary). Besides it can be an interesting tool for guiding evolution of software. Evidently, monitoring is also useful as a mechanism for checking performance, intrusion detection, performance optimization and metering and accounting tasks. Concerning monitoring capabilities, traditional monitoring schemes check actual behaviours against an expected execution model (which can be expressed as rules or policies), but, when properly used in a distributed environment, its potential capability can be substantially increased, such as helping in the discovery of unexpected interactions or model flaws.

Current monitoring architectures are not well suited for highly distributed scenarios and *Cloud Computing* environments (both centralized and decentralized). Moreover, they do not take advantage of the special characteristics these scenarios offer, precisely due to the fact that they are not especially tailored to these scenarios.

The fact that most of the current monitoring languages are based on mathematical (logic or calculus) formalisms represents a drawback with regards to their expressive power and the complexity of the writing monitoring rules. Recently, some have been designed for expressibility, this comes at the price of making the creation of complex specifications a task of epic proportions. Even worse, they are not designed for reducing highly efficient implementations.

In a nutshell, we propose a monitoring model focused on the runtime supervision at several levels (single application instances, a set of different applications, inside the same platform and all the instances of the same application across several platforms) that overcomes the limited efficiency in traditional monitoring systems through a tailored design of a monitoring infrastructure and a monitoring specification language designed. The rationale to produce efficient implementations is that in order to enhance trust in applications, it becomes necessary to analyse their behaviour not only as an isolated unit, but as it is a member of a computing at different ecosystem levels. For this reason, the monitoring architecture that we propose is divided into three levels. These levels allow our monitoring system to improve efficiency and to detect situations not possible with either of these working alone. This means that the results obtained from the analysis at these three levels provides more information than the sum of each one of them working independently, thus allowing the detection of problems not only on the implementations, but also on the models describing them and even problems caused by the interaction of different solutions. This runtime analysis is based on a set of policies (monitoring rules), which are written on a new event-sequence language called EventSteer (detailed in Section 5). As a proof of concept of our model, an actual implementation has been developed for the PASSIVE project.

The remainder of this paper is structured as follows. Section 2 gives a complete background on dynamic verification and system security monitoring. Section 3 presents the main contributions of our approach and describes the dynamic security monitoring and enforcement model. Section 4 deals with prevention through analysis and correlation techniques. Section 5 describes the EventSteer monitoring language. Section 6 describes the strategy to generate finite-state machine as monitoring runtime engines and discusses the performance of the whole system. Section 7 includes a brief description of the implementation of our approach. Finally, Section 8 provides some concluding remarks and discusses future work.

2. BACKGROUND

This section presents the technical background on dynamic security verification (monitoring) including a critical analysis of current research on dynamic verification and on its limitations, especially in relation to the needs of cloud computing applications.

Different approaches have been proposed to provide an assurance of the behaviours of software elements. Among these, we may distinguish between static and dynamic approaches. Static approaches (e.g. code inspection and automated analysis, formal methods, testing, etc.) are based on checking the security of the software before it is actually executed by the real users in their real scenarios. Consequently, static verification activities must be carried out in simulated environments. Dynamic approaches (e.g. monitoring, surveillance and other form of runtime analyses), on the other hand, are based on the observation of the actual behaviour of the software and are carried out in the environment where the software is actually used.

Static approaches can help increase the users' trust in the software. The main advantages of these approaches are that the system verification can be carried out even if the verification processes require complex computations and costly analyses because it is done only once before the software is ever run. This means that the verification time is not an issue. However, they have one major problem, namely, the above-mentioned fact that the results of the static verification do refer to verification activities carried out in a different (frequently highly controlled) environment. The fact that a software element has been formally verified or thoroughly inspected or tested in a laboratory does not mean that the same software will behave correctly in all possible real-world deployment scenarios. Indeed, the influence of the real execution context in the verified software and the interactions with such context cannot be realistically verified with static methods, especially when the heterogeneity and unpredictability of the context are high, as is the case in cloud computing. On the other hand, dynamic verification methods suffer two main problems: in the first place, the complexity and computational power required by these methods are limited because most of the times they must be constrained to avoid that they downgrade the performance of the system. For this reason, the type of analyses done are rather simple, but we must highlight that the complexity of the processes for dynamically verifying a given property of a given software is normally several orders of magnitude simpler than the complexity of statically verifying the same property and system. This claim is easy to justify if we consider that the dynamic verification does only analyse the actual execution trace of the software, while the static verification has to consider all possible execution traces; an additional weakness is the fact that the dynamic verifier must be executed together with the observed software, which means that we introduce a new element that may influence the behaviours that we wanted to observe. On the bright side, if properly designed and implemented, dynamic verification provides a higher assurance level because the analysis of the software behaviour is done in the real execution environment. Consequently, dynamic methods do verify the software in its actual execution context and can potentially discover problems that cannot be observed using static methods.

Due to the nature of our work we will focus in this section on the dynamic verification methods. The dynamic verification of systems has been an active topic in several areas of research including requirements engineering, program verification, safety critical systems and service-centric systems. The traditional way to carry out the dynamic verification is achieved by monitoring the execution of a system and checking its conformity against a set of rules. Such verification requires the following elements, which may be internal (in this case, we

say, we use 'instrumented code') or external to the observed software) [1–4]: (i) a (frequently formal) system model and a specification of a set of targeted safety and security properties (ii) the definition and implementation of methods intended for capturing events of interest and (iii) the deployment of a monitor capable of analysing such events and checking for violations of the rules in order to verify whether the observed behaviour of a system satisfies the required properties.

Also there is an additional element that, though not mandatory for verification, is frequently used in conjunction to dynamic verification models. This is the control/recovery element, which is used by the monitor to react to an improper behaviour. As an example, in aspect-oriented programming (AOP) [5] and monitoring-oriented programming (MoP) [6], the monitor is embedded alongside the instrumented code. In other cases, monitors are independent software modules implemented [3, 7] separately to the system to be observed.

Regarding the languages used for the specification of the system behaviour (monitoring rules), several approaches are based on logics ranging from the popular linear temporal logic (LTL) [8] and variations of it including past and future time LTL (ptLTL and ftLTL, respectively) to more expressive (but less efficient) logics like alternating time epistemic logic (ATEL).

Past and future time LTLs are modal logics for specifying properties of concurrent reactive systems and are used for analysing the traces of execution of such systems. In particular, the Temporal Rover (TR) tool [9] supports a future and past time metric temporal logic (MTL). MTL [10] extends LTL with relative time and real-time constraints. In the context of MoP, any monitoring formalism can be added to the system. ptLTL, ftLTL and extended regular expressions (EREs), which can express patterns in strings in a compact way [11], have been used to formalize properties to be monitored [6]. The proposed algorithms to deal with those specifications use binary transition tree finite-state machines (BTT-FSMs) to monitor ftLTL properties [6], as well as formulas written in a logic based on EREs [11]. Havelund *et al.* [12–14] have developed several algorithms related to temporal logic generation and monitoring, for instance, they propose algorithms for past time logic generation by using dynamic programming [14]. They have also explored the use of the MAUDE rewriting engine [15] for monitoring future time logic [12, 13] and have proposed algorithms that generate Büchi automata adapted to finite trace LTL [16]. Other logic languages used for formalizing properties are EAGLE [1] and HAWK [2].

On the other hand, several calculi, notably event calculus (EC), have also been in the foundation of current monitoring systems. The monitoring and checking (MaC) framework [17] is based on a logic that combines a form of past time LTL and models in real-time via explicit clock variables. JAVA MAC [18] is a prototype implementation of the MaC framework for monitoring and controlling applications written in Java that defines an event-based language to describe monitors. We highlight the fact that in the context of the Java MaC framework, events refer to information that holds instantly during the system runtime, while conditions are defined as illustrating information that holds for a period of time. Mahbub and Spanoudakis [19] have developed a framework for monitoring the behaviour of service-centric systems, which expresses the requirements to be verified against this behaviour in EC [20]. In this framework, EC is used to specify formulas describing behavioural and quality properties of service-centric systems, which are either extracted automatically from the coordination process of said systems (this process is expressed in WS-BPEL) or are provided by the user. In the area of component-based programming, Barnett and Schulte [21] have proposed a framework that uses executable interface specifications and a monitor to check for behavioural equivalence between a component and its interface specification. Robinson [22] has proposed a framework for monitoring requirements based on code instrumentation in which the high-level requirements are expressed in KAOS [23], which is a framework for goal oriented requirements specification based on temporal logic. KAOS has also been used by Feather *et al.* [24] in a framework that was developed to monitor system requirements at runtime while incorporating some capabilities regarding the reconciliation of requirements with the runtime system behaviour. In the field of software systems monitoring, diagnosis focuses on the identification of the potential causes of a system failure. We can consider that the monitoring activity is the process of observing a system in order to detect deviations from normal behaviour and, in particular, violations of the properties required for the system. Such observed deviations are frequently not sufficient for understanding the reasons that underpin the violation of the property and to identify both, the way to avoid that problem in the future (i.e. how to evolve the system) and the way to avoid negative consequences of the detected violation (i.e. how to react to the violation). Consequently, diagnosis is the process of determining the causes of violations detected during the operation of a monitored system. Diagnosis typically involves the identification of the possible trajectories (sequences of events) that have led to a failure. Some proposals use automata (modelling the expected behaviour of a system) to recognize the faulty behaviour [25–29]. In such case, diagnosis is carried out through the synchronization of the automata and the events captured from the actual system. Pencolé and Cordier [27] propose a similar but decentralized approach where synchronization is performed for individual system components and then aggregated for the global system. The problem of fault diagnosis (considering time) has been studied and analysed by Tripakis [29] and Bouyer *et al.* [25], who model the system as a timed automaton. Timed automata extend the finite-state machine models with real-time clocks [30]. References [25, 29] focus on algorithms (diagnosers) that function as efficient online fault detectors of internal faults for any given sequence of observable system-generated events. Tripakis has also worked on the diagnosability of a timed system showing that the problem of checking whether a given timed system

is diagnosable or not is a decidable problem and a diagnoser can be constructed as an online algorithm in the case that the system is actually diagnosable. The algorithm proposed by Tripakis [29] is based on state estimation in order to decide whether a fault has occurred. Due to the high complexity of the previous algorithm, Bouyer *et al.* [25] describe a lower complexity solution using two deterministic timed automata (DTA) for an efficient online diagnosis. Bouyer *et al.* have considered general DTA, as well as a subclass of DTAs called event recording automata (ERA) [30].

2.1. Monitor taxonomy

Monitors and event generators can have different capabilities and structures. This subsection classifies monitoring and event generation capabilities according to three dimensions (i) the controlling capabilities of a monitor, (ii) the time of the event emission with respect to the occurrence of the action described by the event and (iii) the communication type between the monitor and the system.

In the first dimension a distinction is made based on whether the monitor role is to only observe, observe and control or control only.

More precisely: (i) the monitor observes the runtime behaviour of the system by receiving the generated events and it checks whether the monitoring properties hold at runtime (ii) the monitor observes the runtime behaviour of the system by receiving the generated events, it checks whether the monitored properties hold at runtime and forces the system to execute specific actions. These actions can be either preventive or for recovery. This class is also known as closed-loop control and (iii) The monitor forces the system to execute actions without needing to observe the actual state of the system. This class is also known as open-loop control.

The second dimension of the taxonomy presents a distinction according to the time of the event's emission with respect to the occurrence to the event-related action. According to this criterion, we can distinguish between two cases: (i) Emission preceding the action (pre), that is, the event is sent before the action is performed (for example, the event generator sends an event to the monitor to alert that the system wishes to lock a resource before the system actually locks it) and (ii) Emission after the action (for example, the event generator sends an event to the monitor notifying that the system has completed a transaction).

Finally, the third dimension of the taxonomy refers to the type of communication between the monitored system and the monitor. According to this criterion, we distinguish between the following two types of communication: (i) synchronous communication: the event generator uses a blocking send primitive to communicate with the monitor, waiting for a reply from it. This is usually only used when the monitor can exert control over the system and (ii) asynchronous communication: The event generator uses a non-blocking send primitive to communicate with the monitor. It is mainly used when the monitor cannot exert any control over the system or when the control actions can be applied asynchronously. For instance, the monitored system may notify the monitor that it will attempt to perform an action and start performing it without waiting for permission to do so (these are called optimistic transactions). If the monitor subsequently decides that this action is undesirable it can abort or revert the action.

2.2. Security oriented systems

Some of the logics and languages reviewed in the previous section have been used either in their original form or with semantic modifications and extensions to allow the formalization of security properties. Naldurg *et al.* [31], for instance, have proposed a framework for intrusion detection based on the EAGLE language, suitable for expressing temporal patterns that involve reasoning about the data values observed in individual events and thus allowing the description of attacks whose signatures appear to have statistical properties (e.g. password guessing or denial of service attacks). For such attacks there is no clear distinction between an intrusion and normal behaviour, so the intrusion detection involves collecting runtime statistics and using them to evaluate the probability of the occurrence of an attack.

In the area of intrusion detection [32], Ko *et al.* [33] have proposed a specification-based approach, which uses dynamic verification techniques to detect vulnerability exploitations in security-critical programs. According to this framework, it is possible to specify a trace policy that describes the intended program behaviour with regards to security properties. The trace policy then determines security-valid operation sequences of the execution of one or more programs. For specifying such trace policies, Ko *et al.* [33] have developed a grammar, called 'parallel environment grammar (PE-grammar)' whose alphabet consists of system operations. This PE-grammar can express various classes of security trace policies; including behaviour related to system object access, synchronization, sequences of operations and race conditions in concurrent or distributed programs. Schneider [34] has developed a system called execution monitoring (EM) that can monitor violations of security policies by monitoring the system execution steps. This system is based on the security automata of Alpern and Schneider [35], which are a special type of Büchi automata. EM also incorporates mechanisms that can terminate the system execution if it is about to violate its security policy. Following the same automata-based formalism, Ligatti *et al.* [36] extended the control capabilities of security automata by proposing edit automata, which can remove and add letters (i.e. system actions) to the words (i.e. execution traces) they may recognize. Having proposed a security-policy enforcing model that follows the general dynamic verification approach, Bandara *et al.* [37] have specified a language based on EC to model the system behaviour

and to write security policy specifications. The form of EC used in this work was presented in [38] and consists of: (i) a set of time points (that can be mapped to the non-negative integers), (ii) a set of properties that can vary over the lifetime of the system (fluents) and (iii) a set of event types. System operations and domain-independent rules for policy enforcement were specified in this approach using these constructs. According to Bandara *et al.* [37], one can use EC to express system-models containing a combination of authorization, obligation and refrain policies.

Janicke *et al.* [39] have proposed a security model that allows expressing dynamic access control policies, which can be either time or event-driven. A system overall security policy can then be composed out of smaller policies that capture specific requirements and which can be individually verified. The advantage of this access control model is that it allows for expressing both parallel and sequential composition. Janicke *et al.* [39] based their security model on interval temporal logic (ITL), a flexible notation for both propositional and first-order reasoning about intervals of time. ITL allows the expressing of properties for safety, liveness and timeliness. The policy model of Janicke *et al.* [39] provides a wide range of operators (for example, to allow the dynamic addition/deletion of rules or to select different sub-policies based on to the occurrence of an event or a time-out). An important reason for choosing ITL was the availability of an executable subset of the logic, known as Tempura [40]. The use of ITL, together with its subset of Tempura, offers the benefits of traditional proof methods with the speed and convenience of computer-based testing through execution and simulation. Brisset [41] has worked on establishing and ensuring the correct operation of a Java platform security mechanism for runtime authorization of untrusted applications in remote hosts. Sekar *et al.* [42] presented an approach called model-carrying code for mobile code security. Damianou *et al.* [43] have defined a declarative, object-oriented language, called Ponder, to specify security policies that can be monitored and applied at runtime.

2.3. Methods for capturing events

Another important aspect to consider in system monitoring is the mechanism used to retrieve information from the running system. This is normally done based on a series of specific situations (called events) that are relevant for the monitoring activity. Event capturing methods can be divided into external and internal ones.

2.3.1. External event capturing

External methods generate events without altering the code of a system. External event capturing is based on the fact that the software to be monitored has to call the underlying software infrastructure (OS, VM, middleware, etc.) and therefore the event capturing is done by those infrastructures. To do so, these methods extend and/or take advantage of capabilities of the general computational environment in which a system is executed in order to generate the event flow. Reflective middleware approaches [44–46], proxy-based architectures [21] and the use of application programming interfaces [7, 19, 47] constitute examples of methods which belong to this category. Those technologies [48] have been designed to support the development of distributed systems. Capra *et al.* [45] proposed a framework designed to facilitate the adaptation of applications to changing execution conditions. The model considers different layers (operating system, middleware, application and user), each of which is described using metadata and connected by capturing events in order to ease their interaction. Also in this field we can find CARISMA [45] (a context-awareness-based reflective middleware) and XMIDDLE [46]. In the field of component based programming, Barnett and Schulte [21] have proposed a framework that uses executable interface specifications and a monitor to check for behavioural equivalence between a component and its interface specification. The advantages of the external event capturing approach are that it can be applied to any code and that the event capturing can be considered more trusted as it comes from an external (normally trusted) element. On the down side of this approach is the fact that the monitoring rules cannot be tailored to the specific behaviour of the software to be monitored because the events that can be monitored are limited to the calls to the infrastructure. Another drawback is the limited ability to control the monitored software, which is limited to the normal control that the infrastructure can have over the applications running on it.

2.3.2. Internal event capturing

Internal event capturing is based on instrumenting the (source or binary) code in order to insert statements to capture the events of interest and to send them to the monitor. In Robinson [22] the technique of code instrumentation is defined as the insertion of statements into the system's code (source or binary code) for monitoring purposes. Instrumentation can be done manually or automatically. For example, there are tools like Jtrek and JSpy [49] or Joie [50], which automatically instrument Java byte code. During the execution of the instrumented code, an event stream is generated. The generated events can then be passed directly to external monitors or pre-processed before they reach the verification stage. A tool using code instrumentation for monitoring Java-based systems is RMon [22]. In RMon, requirements are initially expressed in the KAOS framework [23], which provides a goal-oriented formal specification language based on temporal logic. Requirements are thus specified as high-level goals which must be achieved by the system. These goals must then be mapped onto low-level events that can be monitored at runtime. The system's code is then instrumented in order to capture these low-level events using the Joie framework [50]. Likewise, in the MONID tool [31], the system-level events are generated by appropriately the instrumented source code.

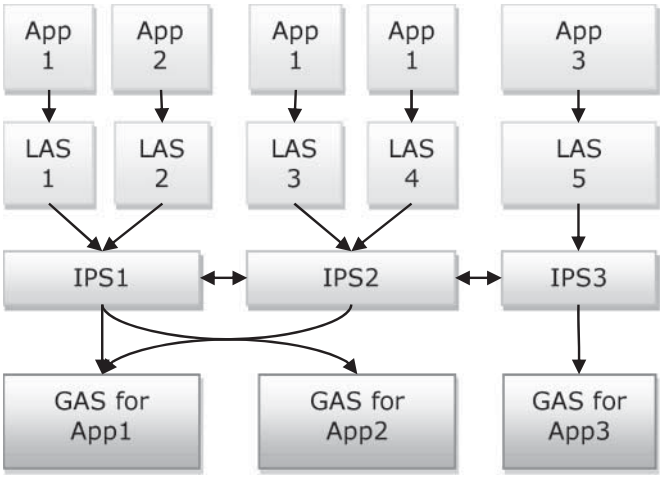


FIGURE 1. Three-dimensional monitoring architecture.

An example of a system that applies instrumentation to the binary code is the Java MaC architecture [18]. In Java, MaC low-level specifications (written in PEDL) are inserted into the byte code of the monitored program through an automatic instrumentation procedure. Some related approaches are AOP [5], which is based on a particular form of code instrumentation to operate. MoP [6], Proof-Carrying Code [51, 52] and Design by Contract [53] are also based on an instrumenting code.

The advantages of the internal event capturing approaches are that the monitored events and thus the monitoring rules can be tailored to perfectly suit the behaviour of the monitored software and the efficiency of the monitoring. The main reason for this efficiency is the fact that the monitoring rules are simpler because they are based on direct observation of the important events, while in the case of external event capturing, the observed events are not the ones that are of direct interest, and rules have to 'deduce' the interesting events from the observed events. The main drawback of this approach is the need to modify the code, but as already mention there are automated tools for this purpose, which facilitate this task.

3. SYSTEM BEHAVIOUR: ENFORCEMENT

In this section, we describe the dynamic security monitoring and enforcement model. As shown in Fig. 1, this infrastructure is based on a three-dimensional model, provided each by the Local Application Surveillance (LAS), the Intra Platform Surveillance (IPS) and the Global Application Surveillance (GAS). Figure 2 shows how the intercommunication between such levels is achieved.

LAS: LAS monitors application instances (so there is one LAS per application instance) and it is the component that most closely resembles the current monitoring systems. Its job is to check whether the application violates any of its established monitoring rules (the rules that express properties that need to be satisfied at runtime) and is used to detect unexpected behaviours, implementation flaws and underpin the trustworthiness of such an application.

The output of the monitoring analysis is then sent to the assigned IPS for further analysis. It is important to take into account that the integration of an LAS into a PASSIVE-virtualized environment does not affect the operation of the virtualized environment itself. The purpose of the LAS is just to provide more information to the LAS administrator about the operation of applications. Using this information, the LAS administrator can modify the application or virtualized environment configuration in order to adapt the system behaviour.

The monitoring infrastructure is divided into the following subcomponents:

(i) *LAS Event Receiver*: Receives the application events and routes them to the analyser.
(ii) *LAS Analyser*: With the events arriving, it proceeds to evaluate whether any of the monitoring rules from the rules databank were violated. The analysis results are expressed in terms of the LAS analyser rule violations, which express abnormal situations.
(iii) *LAS Rules Databank*: Stores the set of the application monitoring rules. This component is managed by the GUI subcomponent.
(iv) *LAS GUI*: A graphical user interface that communicates the LAS administrator with the LAS component. It has these main functionalities: It displays the analysis results from the analyser, manages the monitoring rules in the Databank and inspects the contents of the Event Receiver.

IPS: To deal with potential problems caused by the interaction between different VEs, a second monitoring mechanism is in charge of monitoring at the level of one particular VE. The IPS component serves this purpose by analysing data from different VMs running on the same machine. The IPS component collects information related to the violations of monitoring rules, analyses it and sends the results to the GAS component. The results sent depend on certain rules, so the administrator can choose what information is meant to be sent and what is meant to be kept as confidential.

Specifically, there is one IPS per VM and they are interconnected with other IPS components of the same virtualized environment. They are responsible for analysing the result of the LAS analysers from the same VM, looking for security risks that might arise whenever the different VMs interact as well as whenever different applications from the same VM interact. Selected results of the monitoring analysis are then sent to the different GAS components (assuming such a GAS is available) for further analysis.

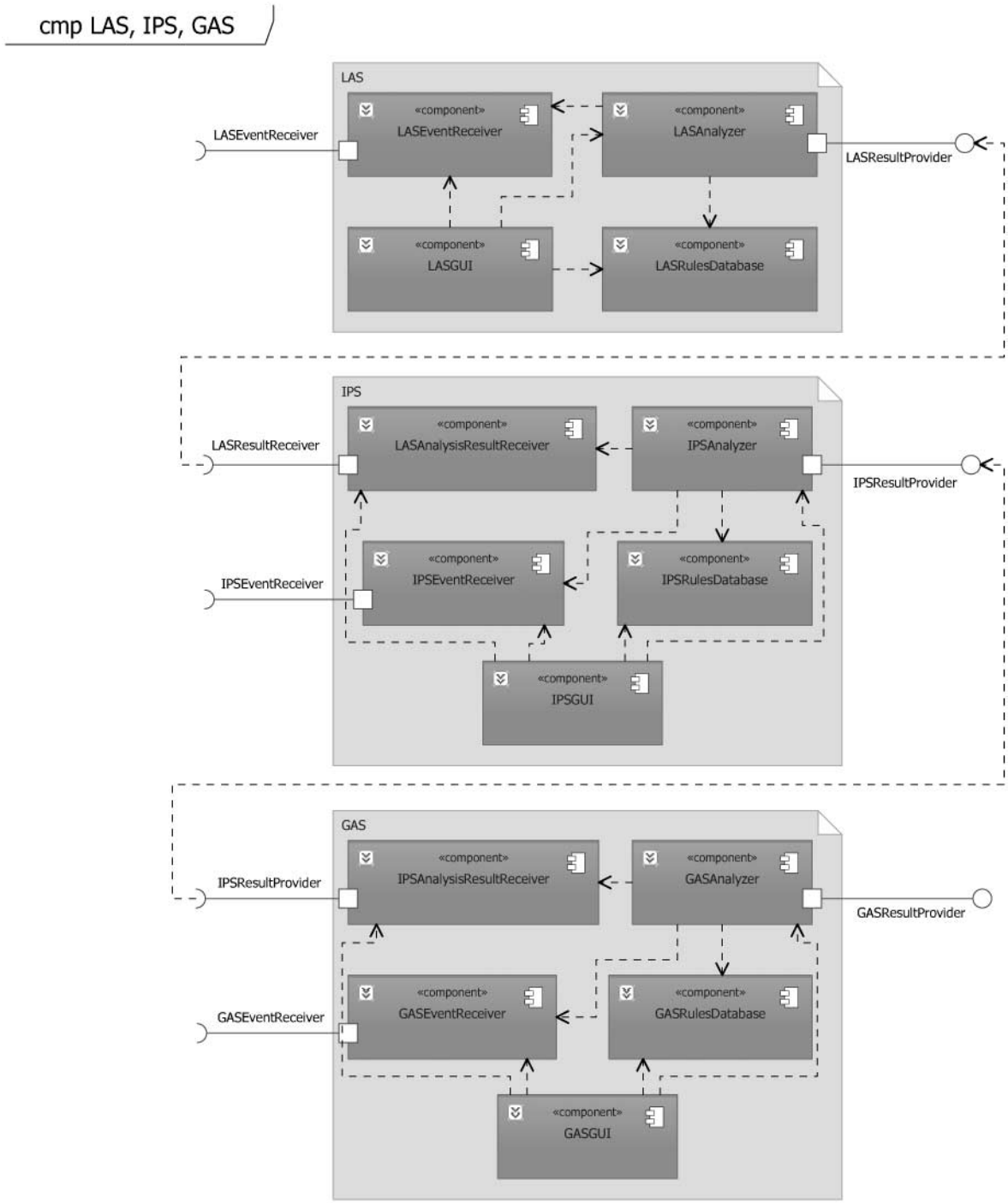


FIGURE 2. Intercommunication between different levels.

Likewise, the integration of an IPS into a PASSIVE virtualized environment does not affect the operation of the virtualized environment itself. The purpose of the IPS is just to provide more information to the IPS administrator about the operation of applications. Using this information the IPS administrator can modify the application or virtualized environment configuration in order to adapt the system behaviour. IPS is divided into the following subcomponents:

(i) *IPS Event Receiver*: Receives the external events and routes them to the analyser.

(ii) *LAS Analysis Result Reader*: Reads the result of the analysis performed by the different LAS components running on the same VM.
(iii) *IPS Analyser*: With the LAS Analysis results, it proceeds to evaluate whether any of the rules from the rules databank have been violated. The analysis results are expressed in terms of IPS analyser rule violations, which express abnormal situations.
(iv) *IPS Rules Databank*: Stores the set of the intra-platform monitoring rules. This component is managed by the GUI subcomponent.
(v) *IPS GUI*: A graphical user interface that communicates the IPS administrator with the LAS component, with these main functionalities; it displays the analysis results from the analyser, manages the monitoring rules in the Databank and inspects the contents of the LAS Analysis Result Reader.

GAS: To support monitoring of specific software pieces and detecting problems with non-compliant implementations (as well as problems in the modelling), the GAS components perform vertical analysis. They analyse data from different machines referred to the same software (application). Such GAS components receive information from several IPS components and perform a new analysis on it. Thus, the GAS components have a global view of what is the behaviour of the software in different virtualized environments from different machines, and thus is able to deduce proper conclusions. The existence of GAS components benefits both users of the applications and applications developers.

This level is optional and there is one GAS per application (not instance). Furthermore, they might reside well outside the virtualized environment if necessary. Their task is a secondary form of analysis of the result of the IPS analysers from all VMs in all virtualized environments in order to be able to detect the application global design flaws, making it an invaluable resource for the developers of such applications. GAS is composed of the following subcomponents:

(i) *GAS Event Receiver*: Receives the external events and routes them to the analyser.
(ii) *IPS Analysis Result Reader*: Reads the result of the analysis performed by the different IPS components located globally.
(iii) *GAS Analyser*: With the IPS Analysis results, it proceeds to evaluate whether any of the rules from the rules databank have been violated. The results of the analysis are expressed in terms of GAS analyser rule violations, which express abnormal situations.
(iv) *GAS Rules Databank*: Stores the set of the intra-platform monitoring rules. This component is managed by the GUI subcomponent.
(v) *GAS GUI*: A graphical user interface that communicates the GAS administrator with the GAS component. It has these main functionalities. It displays the analysis results from the analyser, it manages the monitoring rules in the Databank and it inspects the contents of the IPS Analysis Result Reader.

All these systems help achieve our goal of increasing the security and reliability of virtualized environments and, therefore, cloud computing by:

(i) Easing the identification of the origin of errors (thanks to the LAS, IPS and GAS components, which are able to monitor each application separately or as a whole).
(ii) Capturing precise and specific information on attacks, errors and malfunctioning.
(iii) Lowering the time required to identify and fix errors (a good set of monitoring rules helps with error identification before it further propagates in the application and becomes harder to track).
(iv) Early detection (the monitor provides the capability to inform the developer of an unexpected behaviour immediately after the first case happening).
(v) Increasing the protection of the VE code during the monitoring process (by creating and using custom system monitoring rules).
(vi) Increasing the ability to assess the integrity and compliance of the VE after monitoring (by being able to check at any time the current state of monitoring rules).

4. PREVENTION: ANALYSIS AND CORRELATION

Runtime software monitoring of security properties is widely accepted as a way to increase system resilience to dependability failures and security attacks. Proposed models of monitoring advocate the need for this form of system verification and the development of a monitoring framework that supports it. It should be noted, however, that whilst current monitoring systems are able to detect violations of S&D properties at runtime, they cannot always provide the necessary information for understanding the reasons that triggered the violation of an S&D property and therefore are not prepared to decide response to said violation.

Furthermore, it is often necessary to try to predict a violation before it even happens by using the available current system-state information rather than wait until all the information to make a final decision becomes available. This is because an accurate early prediction can widen the scope of possible reactions to the violation or even provide scope for taking pre-emptive action that prevents the violation.

In our monitoring system, the absence of a signal after the elapse of a given signalling period can be detected by specifying a monitoring rule, requiring that the time between two consecutive signals from the same source (e.g. an application or a device) should not exceed the given period. Detecting, however, the occurrence of a violation of this rule is not in itself

sufficient for establishing the reasons why a source has failed to send the expected signals. In such cases, a further search for possible causes of the violation are useful when deciding how to react to the violation.

As an example, consider that the violation might have been caused because the source is malfunctioning and has stopped sending signals after a certain point in time; the source involved is no longer present in the area covered by the server; some of the signals sent by the source have been lost in the communication channel between the source and the server; and the signal that was used to determine the start of the last period of checking was sent by an external agent (attacker) who managed to fake the identity of the source (i.e. an attacker).

Although the preceding list of possible causes is not exhaustive, it demonstrates that a decision about what would be an appropriate reaction to the violation depends on the reason(s) that have caused it and, therefore, the selection of the appropriate responding action cannot be made solely on the basis of knowledge of the violation but requires additional diagnostic information. The diagnosis mechanism of PASSIVE is invoked after the detection of the violation of a monitoring rule in order to find possible explanations for the reasons underpinning the occurrence of the events involved in the violation of the rule and assess their genuineness.

This mechanism produces diagnostic information through a process of four stages described in the following paragraph. In the generation stage, the diagnosis mechanism generates all the possible explanations of the events, which are involved in the violation. These explanations are generated using an abductive reasoning based on assumptions about the behaviour of the components of the system. Application vendors provide these assumptions. In the effect identification stage, the diagnosis mechanism derives the possible consequences (effects) of the potential explanations that were identified in the previous stage. The consequences are generated from the abducted explanations and system assumptions using deductive reasoning. In the plausibility assessment stage, the diagnosis mechanism checks the expected effects of explanations against the event log to see whether there are events that match them or, equivalently, the existence of further supportive evidence for the explanation. In the diagnosis generation stage, the diagnosis mechanism produces an overall diagnosis for the violation including belief measures in the genuineness of the events involved and the most plausible explanations that have been identified for these events (if any).

5. MONITORING SPECIFICATION AND ITS LANGUAGE

Any monitoring system is based on a set of well-defined behaviour policies, also known as monitoring rules. In our particular monitoring system such rules are part of the monitoring specifications, which are written in a new language called EventSteer.

EventSteer is an extended event-sequence language (namely, a language that allows the user to create rules based on an expected or unexpected flow of events). Like EC [44] it uses two basic concepts for representing the properties of systems that change over time; events and fluents. Whilst an event is defined as something that occurs at a specific moment in time and has instantaneous duration, a fluent is a condition that has different values at different moments in time. It is also extended because it does not only allow for specifying event sequences but also the possible consequences of failing to validate such sequences in a standard imperative programming syntax (Java in our case, but could be any other).

The main rationale behind the creation of a new language is based on three key points. The first is that while EC provides an elegant mathematical way to formulate specifications, sometimes they are too complex to be understood at first glance. The second is that the free mixing of time and events is difficult to compile, interpret and debug due to the lack of a given fixed sequence to try to follow. The third and last one is that this very same lack of a fixed sequence in time means that it is extremely complicated to devise an optimal way to evaluate these rules at runtime.

Because of all this, one of the initial design goals of EventSteer was to overcome the limitations and drawbacks of current EC-based approaches. In particular, we focused, on the one hand, facilitating the expression of time, by avoiding mixing time and event processing, and by providing a clear distinction between the settings of a time variable (i.e. assigning a value to the variable) and the use of it (i.e. using the assigned value in the evaluation of an expression). On the other hand, we also focused on expressing time in a past to future fixed sequence that facilitates the creation and optimization of the implementation of those rule checkers (in particular, by using FSMs).

A complete description of the language is beyond the scope of this paper. Nevertheless a description of the basic elements that are the basis for the rationale of the project has been included as follows.

Variables: In *EventSteer*, variables (also known as fluents in EC) can have different values in different moments in time (as their name indicates). They have names for identification purposes and can be of one of the several types (Boolean, integer or double). Variables are set to specific values in the initialization section as well as in the consequences of monitoring rules as will be described in the Consequences section. Likewise, variables can be used in guards as part of event expressions by using them inside a Boolean expression that establishes when a given event can be accepted or not.

Time: Our notion of time is based on LTL [45]. LTL is a modal temporal logic with modalities referring to time that allows the definition of expressions about the future such as the fact that a condition will become true in the future, that it will be true until something happens, etc.

Opposed to other temporal logics such as computation tree logic (CTL) or ATEL, which allow the expression of different possible paths into the future, LTL can only express conditions on one path (i.e. it implicitly quantifies universally over paths), hence the name linear. This is a limitation when expressing the behaviour of software or hardware elements, especially for verifying safety or liveliness properties using model checkers. However, for the purposes of monitoring, the only path that we need to analyse is the actual execution. Moreover, as we will show, in our language, different rules can represent overlapping paths, and there are several constructs to facilitate the expression of complex rules. Therefore, the limitation of representing one path in a rule does not constitute a problem for us. For this reason, it is possible to represent all monitoring rules using LTL.

Following the standard EC approach described in [30] the time type is considered to have discrete values and consists of a set of ordered time points that can be mapped to the set of non-negative integers. However, as has already been mentioned, the treatment of time has been carefully considered in EventSteer and there are two main aspects to highlight: on the one hand, time treatment is detached from event treatment and, on the other hand, there is a clear distinction between the setting of time variables and the use of these variables once they have been instantiated.

Here is the treatment of time in EventSteer in more detail:

(i) Time variables and deadlines are linked to event expressions. They are enclosed in brackets after them as follows: eventExpression timeVariable. The semantics of this construct is that timeVariable is set to the actual time when the eventExpression is verified.
(ii) Time variables are used in timeExpressions and can be part of time ranges. Its syntax is timeVariable (+timeConstant). The optional integer constant is expressed in msecs.
(iii) Time ranges, used in deadlines, take the form [(minTimeExpression)…(maxTimeExpression)]. Note that if the optional minTimeExpression is not written it means the minimum time range is zero. Also if the optional maxTimeExpression is not written it means the maximum time range is infinite.

Events: Events are the central element of any monitoring language. Events represent the connection between the monitor and the monitored system. For the sake of the current discussion and throughout the paper, we will consider that events are instantaneous situations that indicate some relevant change in the monitored system.

In our framework, we consider three kinds of events; namely external, virtual and special events.

External events: External events are externally generated events that are triggered at certain arbitrary points by an external code. Examples of external events might be access to a file, the beginning of a password request, the acceptance of said password or even a certain button being pressed. In our framework, we concentrate on events that are relevant for the security of the system, but the EventSeer language is of course independent of this. External events are expressed in our language by directly using the event name and its attributes as parameters if applicable. This is, eventName(attrib1, attrib2, etc.).

Virtual events: Virtual events are monitoring oriented, internally triggered events we use in order to keep control of the system state. Note here that by system state we really mean the monitoring machine state or whatever underlying implementation might be using the language. These kinds of events are user-defined internal events. These events are sent inside the consequence code, thanks to an special method named SendVirtualEvent(evName, evParameters).

Special events: Special events are event predicates with special meanings. They are:

(i) *true*: An event that is completed instantly, i.e. as soon as it is needed.
(ii) *false*: An event that is never completed and fails the completion of the event expression instantly.

Sequences: Sequences are the way to relate events in time. They define a series of temporally ordered event expressions and are done thanks to the sequence operators ',' and '->'. Other available operators are the complement '?' operator, the any '−' operator, the all '+' operator and the except if '!' operator.

A short explanation of each of those operators follows:

(i) *',' Witnessed sequence*: It defines what previous event expressions need to be completed in order to advance the sequence. This, like most other operators, is 'open', meaning that no other events can arrive in between without making the sequence fail.
(ii) *'->' Rule expected sequence*: It works like the witnessed sequence operator, with the only difference being after it is used if the sequence fails the rule will become violated and will trigger its consequence.
(iii) *'−' Any*: This operator is used to define an event expression constituted by several alternative sequences. Namely, as long as one of all the event expressions is verified the combined event expression is considered to be verified.
(iv) *'+' All (in any order)*: This operator defines a sequence that is verified when all the event expressions that it contains are verified regardless of the order in which such event expressions happen.
(v) *' ' ' Complement*: This operator is used to express the set of events that are not to be considered in a certain context.
(vi) *'!' Except if*: This operator is used to express the set of events that are not to be witnessed in a certain context.

Reactions: Reactions allow the monitoring machine to react to certain event expressions by making changes to the internal monitoring machine state (variables/fluents, sending other kinds of virtual events etc.). In EventSteer, reactions are stated as rules without a '->' operator. For instance, there could be a reaction to increase an nLoginAttempts integer variable each time an event warning of a failed login attempt was produced.

Consequences: Consequences are the actions that must be triggered when a rule is violated and can have a direct effect on the monitored system or its internals. Examples of consequences can be restarting the system, halting it, suggesting actions to a higher controller or changing a system parameter, as well as changing variables, etc.

Real life example: This example shows the applicability of this language to a given real life case, which, with a bit of intuition on the reader's part, might prove to be useful for a basic understanding of the language.

Let us imagine a computer system which has the following set of requirements:

(i) When a user fails to login three times in a row, a login error screen must appear. If for some reason it does not, then the system will be halted.
(ii) Users can only transfer data if they are logged in and such a transfer can only take 20 s at most to complete.

```
events {
  evLoginAttemptFailed, evLoginError,
  evLoggedIn, evLoggedOut,
  evTransferStarted, evTransferCompleted
}

variables {
  int nLoginAttempts, bool bLoggedIn
}

init {?
  bLoggedIn = false;
  nLoginAttempts = 0;
?}

rules {
  // REACTIONS //
  // if a login failed the number of login
  // attempts increases
  rLoginFailed: evLoginAttemptFailed;
  {? nLoginAttempts++; ?}

  // if a login happened then we are logged in
  rLoggedIn: evLoggedIn;
  {?
        bLoggedIn = true;
        nLoginAttempts = 0;
  ?}

  // if a login failed then we are logged out
  reactionLoggedOut: evLoggedOut;
  {?
        bLoggedIn = false;
  ?}

  // RULES //
  // if a login failed when the number of
  // attempts was >= 3 then we must- see
  // a login error
  rLoginAttempts:
  evLoginAttemptFailed / nLoginAttempts >= 3
        -> evLoginError;
  {?
        SystemHalt();
  ?}

  // if a transfer started then we must-
  // be logged in
  rTransferLoggedIn:
  evTransferStarted -> true / bLoggedIn;
  {?
        SystemHalt();
  ?}

  // if a transfer started at time t1 then
  // it must- be completed before t1+20000 msecs
  rTransferCompletion:
  evTransferStarted {t1}
        -> evTransferCompleted [..t1+20000];
  {?
        SystemLog();
  ?}
}
```

6. FINITE-STATE MACHINE GENERATION STRATEGY

We have introduced a global overview of our dynamic monitoring system and the description of the monitoring specification language to process the events in rules. As we have previously mentioned, the real-life usability of a monitoring system in most cases depends heavily on its impact on performance. In order to address this problem, we decided to make use of a strategy based on the generation and use of FSMs as the underlying monitoring runtime engine.

While one of the strengths of EC-based formal languages is the ability to write non-time-linear sequences, it happens to also be its major weakness when one of the goals is performance. In other words, this expressive power is paid in full in the end by the compiler of these languages, which need to generate extremely complex verification code (runtime engines), obviously leading to performance bottlenecks (a tremendously important aspect for real-life usage scenarios).

A specific example would be the fact that in EC-based languages it is possible to write an expression such as: 'Event A has to be observed; then another event B has to be observed not later than 500 ms after A has been observed; and a C has to be observed somewhere in the middle of A and B'. Although it is a perfectly valid expression, the fact that it is possible to write it out of sequence (time-wise), among other things, either requires a very CPU intensive runtime verification of such expression or creates the need of some set of really

effective compiler optimizations that are not always feasible. Moreover, due to this flexibility (lack of constraints) in the expressive power, it is harder to determine the soundness of the specifications and, therefore, mistakes are both easier to make and harder to find. The above considerations constitute one of the main reasons for the addition of a strict time-linear sequence structure into our language (EventSteer), a feature that makes it possible, and even relatively easy, to generate performance-optimized FSMs specially tailored to efficiently check EventSteer specifications.

All of this is one of the main reasons for the addition of a strict time-linear sequence structure into our language, a feature that makes it possible, and even relatively easy, to generate performance-optimized FSMs specially tailored to such expression testing and verification.

Traditional methods to process the relationship between events are rather limiting because its expressive power allows non-linear sequences in time, i.e. formal and formal languages based on the EC rules allow writing without regard to a predetermined linear order, for example, may be a situation in which an event A has to be observed before time 100, another event B before a time event 500 and a C before time 300. Due to the nature of the EC, the generated code can verify these expressions is extremely complex and this leads to a clear inefficiency in terms of computational speed. For this reason we have introduced a clear language restrictions in terms of linear time, it has to follow a set sequence and clear in time. And thanks to this feature is possible to generate FSMs optimized for the testing of such expressions in terms of efficiency. Along this section is described how the transformation from the event-sequence language to the FSMs is achieved.

6.1. Language elements to sub-state machines transformation

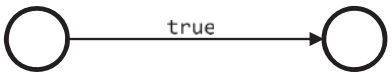
The transformation is based on the fact that every event-sequence language element can be transformed into sub-state machines (SSMs), which are just little FSMs, part of the complete resulting FSM. These transformations are unique for each element, since they are related to the particular properties of every one of them. Among the possible transformations, there are:

6.1.1. Atomic

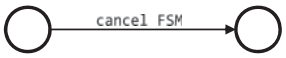
We have defined three different atomic operations:*ev(filters), true* and *false*.

ev(filters): Create a start and end state with a transition with the given filtered event.

true: Create a start and end state with a transition that always triggers.



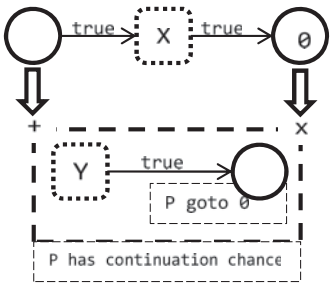
false: Create a start and failure end state with a transition that always triggers.



6.1.2. Operators

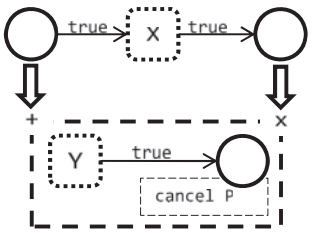
LSEQ , RSEQ and *LSEQ ->RSEQ*: Merge the last state of LSEQ with the first state of RSEQ.

LSEQ − RSEQ: Merge the first states and last states of LSEQ and RSEQ.



$SEQ + SEQ + \ldots$: Since it is possible to translate this operator to a series of finite permutations of the ',' and '—' operators, the previous transformations are applicable.

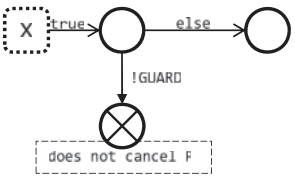
LSEQ ! RSEQ: On the first state of LSEQ spawn a parallel SSM that ends in failure and dispose of it in the last state of LSEQ.



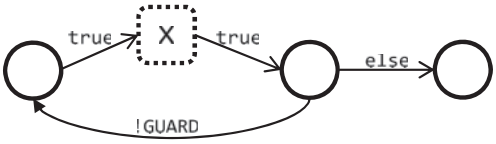
6.1.3. Guards

SEQ/GUARD: Add to the last state of SEQ a new guard immediate transition (transition that is checked only once on

entry and before any non-immediate ones) that leads to a failure state.

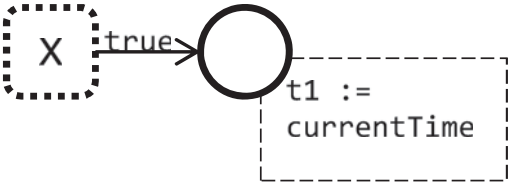


SEQ ^ GUARD: Add to the last state of SEQ a new guard immediate transition (transition that is checked only once on entry and before any non-immediate ones) that leads to the first state of SEQ.

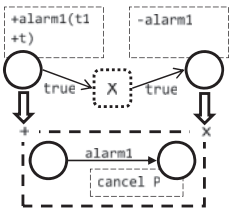


6.1.4. Time

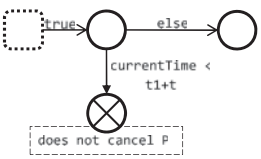
SEQ {t1}: Add to the last state of Left an entry action that sets the time variable to the current time.



SEQ [..t1+x]: Adds to the first state of SEQ an entry action that sets an alarm at t1+x and makes the first state of Left spawn a parallel SSM to listen for that alarm. That SSM is then disposed in the last state of SEQ.



SEQ [t1+x..]: Add to the last state of SEQ a new time immediate transition (transition that is checked only once on entry and before any non-immediate ones) that leads to a failure state.

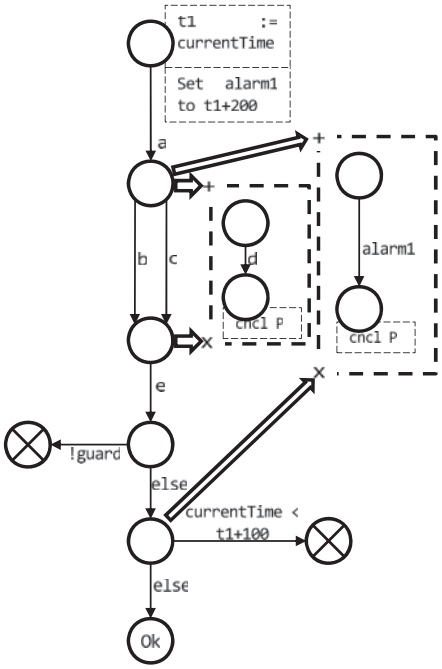


6.1.5. Sample

Finally, here is a complete example of the resulting transformation from an event sequence to an FSM.

```
a {t1}, ((b | c) ! d, e / guard)

    [t1+100..t1+200]
```



7. IMPLEMENTATION

The PASSIVE project has developed an improved security model for virtualized systems that tries to solve, specifically, the problems related to the management of large applications running on virtualized platforms in e-Government scenarios. One of the main concerns raised by the shared-resource nature of virtualization technologies is the data confidentiality conflict, therefore this new security model will ensure: (i)An adequate separation of concerns (e.g. policing, judiciary) (ii) That threats from co-hosted OS s can be detected and dealt with and (iii) that public trust in application providers can be maintained even when the underlying host or any other hosted guest's change. To achieve these aims, PASSIVE offers: (i) A policy-based Security architecture that allows security provisions to be easily specified and efficiently addressed; (ii) Fully virtualized resource access with fine-grained control over device access, running on an ultra-lightweight Virtual Machine Manager; and (iii) a lightweight, dynamic system for host and application authentication in a virtualized environment. Thanks to these elements PASSIVE lowers the barriers to adoption of virtualized hosting by government users, leading to considerable gains in

energy efficiency, reduced capital expenditure and adding the flexibility offered by virtualization.

As a proof of concept of our model, an actual implementation has been developed for the PASSIVE project. The PASSIVE monitoring engine is in charge of observing specific events coming from the VMs and matching these events to monitoring rules written in a new monitoring language (EventSteer). In the framework of PASSIVE, such monitoring specifications are specially tailored to ensure that specific security policies are enforced. That is, monitoring is directed by global security policies. However, in other scenarios, both the proposed architecture and the language can be applied to monitor any arbitrary behaviour of the monitored software. In PASSIVE, when a deviation from the acceptable policies is detected, the monitoring component communicates any relevant data to the PASSIVE policy decision point, which chooses the proper reaction. Our monitoring architecture provides advanced monitoring capabilities based on several levels of monitoring: (1) individual application instances, (2) the set of different applications running on the same platform and (3) different instances of the same application across several platforms. Levels 1 and 2 are performed in the same computing platform of the monitored application, while level 3 is performed externally. There is a monitoring element responsible for each layer. LAS deals with level 1, IPS with level 2 and GAS with level 3. Additionally, to the provision of capabilities that are not possible with traditional monolithic approaches, the architecture also improves the efficiency of the monitoring system because the monitoring effort is divided across platforms (each platform is only concerned with the monitoring activity that is relevant for it). In particular, the monitoring done by the LAS and IPS components are simplified because they are relieved of all evolution-oriented monitoring activities, which are very expensive in computational terms, especially when done locally. In fact, by moving the GAS component out of the computing platform and allowing it to control different instances, the efficiency of the level 3 monitoring is also improved. Finally, it is important to note that the externalization of the GAS might introduce privacy issues and consequently we have added privacy control capabilities to the IPS to solve these issues.

8. CONCLUDING REMARKS AND FUTURE WORK

As highly distributed implementation wise systems (such as cloud computing and virtualized environments) become more and more common each day, security is quickly becoming both a key point of concern and of research. One of the main reasons for the concern is the fact that, due to the intrinsic properties of such systems, (i) many programs from many different developers might be sharing the very same hardware and (ii) most of them are basically exposed to the internet as a whole, opening the door to many security threats both old (e.g. SQL injections) and brand new (e.g. espionage of other programs running on the same hardware).

While the old threats can be pretty much covered by standard systems, the new ones cannot be easily detected by them, thus opening the door for better specially tailored tools. That being said, we hope that our monitoring research will be one of many tools to come in the future with the same focus namely to solve this special kind of new security problems.

However, a high-level architecture must also reside over a solid low-level base. We think that both the new event-sequence language and its finite-state machine compilation serve that purpose well. Thanks to the new even-sequence language we are able to observe as well as control any layer of the architecture (be it with monitoring specifications for the LAS, IPS or GAS level) and thanks to the finite-state machine implementation it is fast enough not to tax the system resources to a level that would be impractical performance-wise.

Summarizing, as we have shown throughout the paper, our scheme allows:

(i) A multi-layer monitoring architecture that provides new capabilities for highly distributed implementation wise systems such as cloud computing.
(ii) A new event-sequence language able to express monitoring specifications which allows for observation as well as control.
(iii) A fast implementation based on finite-state machines adequate for real life performance-efficient scenarios.

As for immediate future work we are targeting, first, a monitoring specification validation system which will make use of sets of emulated event flows that will allow third parties to test the sturdiness and good behaviour of their specifications before the actual programs can generate such events. This would allow a more loosely coupled approach between the team making such specifications and any other team making the actual programs.

Also we are targeting the optimization of the resulting FSMs in order to further improve the usability in performance-efficient scenarios with complex sets of specifications, such as the case of cloud computing.

We are also studying language extensions to further improve expressibility given that we can find use cases which would not be presently addressable.

As to the not so immediate future we are targeting the possibility of an automated system evolution driven by the analysis results from the monitoring subsystem, which would allow the automatic patching of security flaws as soon as they are detected by checking on-line whether new component versions that address the violation of certain rules are available and updating them.

Also we are considering a component capable of monitoring operating system events instead of instrumented monitoring, thus allowing the system to actually enforce and check both system-wide or wide policies.

ACKNOWLEDGEMENTS

Authors would like to thank Prof George Spanoudakis for his comments and advice on the initial versions of this work.

FUNDING

This research was undertaken as part of the EC Framework Programme as part of the ICT PASSIVE project (http://ictpassive.eu).

REFERENCES

[1] Barringer, H., Goldberg, A., Havelund, K. and Sen, K. (2004) Rule-Based Runtime Verification. *5th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, LNCS 2937, Springer, pp. 44–57.

[2] d'Amorim, M. and Havelund, K. (2005) Event-based Runtime Verification of Java Programs. *Proc. 3rd Int. Workshop on Dynamic Analysis*, St. Louis, MO. pp. 1–7. ACM Press, NY.

[3] Havelund, K. and Rosu, G. (2004) An overview of the runtime verification tool Java PathExplorer. *Methods in System Design*, **24**, 189–215.

[4] Spanoudakis, G. and Mahub, K. (2006) Non-intrusive monitoring of service based systems. *Int. J. Coop. Inf. Syst.*, **15**, 325–358.

[5] Kiczales, G. and Lampig, J. (1997) Aspect-Oriented Programming. In Aksit, M. and Matsuika, S. (eds) Lecture Notes in Computer Science 1241, pp. 220–242. Xerox Palo Alto Research Center.

[6] Chen, F. and Rosu. (2003) *Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation*. Electronic Notes in Theoretical Computer Science 89, No. 2, Elsevier.

[7] Artho, C., Schuppan, V., Biere, A., Eugster, P., Baur, M. and Zweimuller, B. (2004) JNuke: Efficient Dynamic Analysis for Java. *Proc. 16th Int. Conf. Computer Aided Verification*, Boston, MA. LNCS 3114, pp. 462–465. Springer–Verlag.

[8] Pnueli. A. (1977) The Temporal Logic of Programs. *Proc. 18th IEEE Symp. Foundations of Computer Science*, Los Alamitos, CA. pp. 46–77. IEEE Computer Society.

[9] Drusinsky, D. (2000) The Temporal Rover and the ATG Rover. In Havelund, K., Penix, J. and Visser, W. (eds) *SPIN Model Checking and Software Verification*, LNCS 1885, pp. 323–330. Springer.

[10] Chang, E., Pnueli, A. and Manna, Z. (1994) Compositional Verification of Real-Time Systems. *Proc. 9th IEEE Symp. Logic in Computer Science*, Paris (France). pp. 458–465. IEEE Computer Society Press.

[11] Sen, K. and Rosu, G. (2003) Generating Optimal Monitors for Extended Regular Expressions. *Proc. 3rd Int. Workshop on Runtim Verification*, Boulder, CO, USA. pp. 162–181. Elsevier Science.

[12] Havelund, K. and Rosu, G. (2001) Monitoring Java Programs with Java PathExplorer. *1st Int. Workshop on Runtime Verification*, Paris (France). pp. 97–114. Elsevier Science.

[13] Havelund, K. and Rosu, G. (2001). Monitoring Programs using Rewriting. *Proc. Int. Conf. on Automated Software Engineering (ASE'01)*, San Diego, USA. pp. 135–143. Institute of Electrical and Electronics Engineers. Institue of Electrical and Electronics Engineers.

[14] Havelund, K. and Rosu, G. (2002) Synthesizing Monitors for Safety Properties. *Tools and Algorithm for Construction and Analysis of Systems*, Grenoble (France). LNCS 2280. Springer–Verlag.

[15] Clavel, M., Durán, F.J., Eker, S., Lincoln Martí-Oliet, N., Meseguer, J. and Quesada, K.F. (1999) The Maude System. *Proc. 10th Int. Conf. on Rewriting Techniques*. Int. Conf. on Parallel and Distributed Processing Techniques and Applications, Las Vegas (USA).

[16] Giannakopoulou, D. and Havelund, K. (2001) Automata-Based Verification of Temporal Properties on Running Programs. *Proc. Int. Conf. on Automated Software Engineering*, Coronado Island, CA. pp. 412–416. ENTCS. IEEE Computer Society.

[17] Lee, I., Kannan, S., Kim, M., Sokolsky, O. and Viswanathan, M. (1999) Runtime Assurance Based on Formal Specifications. *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications*. Las Vegas (USA).

[18] Kim, M., Kannan, S., Lee, I., Sokolsky, O. and Viswanathan, M. (2001) Java-mac: A Runtime Assurance Tool for for Java Programs. *Electronic Notes in Theoretical Computer Science*, Vol. 55. Elsevier Science Publishers.

[19] Mahub, K. and Spanoudakis, G. (2004) A Framework for Requirements Monitoring of Service Based Systems. *Proc. 2nd Int. Conf. on Service Oriented Computing*, New York (USA).

[20] Shanahan, M. (1999) The Event Calculus Explained. *Artificial Intelligence Today*, LNAI 1600, pp. 409–430. Springer–Verlag.

[21] Barnett, M. and Schulte, W. (2001) Spying on Components: A Runtime Verification Technique. *Workshop on Specification and Verification of Component Based Systems*, Tampa (USA).

[22] Robinson. W. (2002) Monitoring Software Requirements using Instrumented Code. *Proc. Hawaii Int. Conf. on Systems Sciences*, Big Island (Hawaii).

[23] Dardenne, A., van Lamsweerde, A. and Fickas, S. (1993) Goal-directed requirements acquisition. *Sci. Comput. Program.*, **20**, pp. 3–50.

[24] Feather, M.S., Fickas, S., van Lamsweerde, A. and Ponsard, C. (1998) Reconciling System Requirements and Runtime Behaviour. *Proc. 9th Int. Work. on Software Specification & Design*, Mie (Japan).

[25] Bouyer, P., Chevalier, F. and D'Souza, D. (2005) Fault Diagnosis using Timed Automata. *Proc. 8th Int. Conf. on Foundations of Software Science and Computations Structures*, Edinburg (UK). LNCS 3441, pp. 219–233. Springer.

[26] Grastien, A., Cordier, M. and Largouët, C. (2005) Incremental Diagnosis of Discrete-Event Systems. On Principles of Diagnosis (DX05).

[27] Pencolé, Y. and Cordier, M. (2005) A formal framework for the decentralised diagnosis of large scale discrete event systems & its application to telecommunication networks. *Artif. Intell.*, **164**, 121–180.

[28] Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K. and Teneketzis, D.C. (1996) Failure Diagnosis using Dicrete-Event Models. *IEEE Trans. Control Syst. Technol.*, **4**, 105–124.

[29] Tripakis, S. (2002) Fault Diagnosis for Timed Automata. *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant*

Systems, Oldenburg (Germany). LNCS 2469, pp. 205–224. Springer.

[30] Alur, R., Fix, L. and Henziger, T.A. (1994) A Determinizable Class of Timed Automata. *Proc. 6th Conf. on Computer Aided Verification*, Stanford (USA). LNCS 818, pp. 1–13. Springer.

[31] Naldurg, P., Sen, K. and Thati, P. (2004) A Temporal Logic Based Framework to Intrusion Detection. *Proc. Int. Conf. on Formal Techniques for Networked and Distributed Systems*, Madrid (Spain).

[32] Lazarevic, A., Kumar, V. and Srivastava, J. (2005) Intrusion Detection: A Survey. *Managing Cyber-Threats: Issues Approaches & challenges*. Springer.

[33] Ko, C., Ruschitzka, M. and Levitt, K. (1997) Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach. *IEEE Symp. on Security and Privacy*, Oakland (USA). pp. 175–187. IEEE Society Press.

[34] Schneider, F.B. (1998) Enforceable Security Policies. Cornell University Technical Report TR98-1664.

[35] Alpern, B. and Schneider, F.B. (1987) Recognizing safety and liveness. *Distrib. Comput.*, **2**, 117–126.

[36] Ligatti, J., Bauer, L. and Walker, D. (2005) Edit Automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Secur.*, **4**, 2–16.

[37] Bandara, A.K., Lupu, E.C. and Russo, A. (2003) Using Event Calculus to Formalise Policy Specification and Analysis. *Proc. Policies for Distributed Systems and Networks*, POLICY, Lake Como (Italy). pp. 26–39. IEEE Computer Society.

[38] Russo, A., Miller, A., Nuseibeh, B., and Kramer, J. (2002) An Abductive Approach for Analysing Event-Based Requirements Specifications. *Presented at 18th Int. Conf. on Logic Programming*, Copenhaguen (Denmark).

[39] Janicke, H., Siewe, K., Jones, F., Cau, A. and Zedan, H. (2005) Analysis and Run-time Verification of Dynamic Security Policies. *Workshop on Defense Applications of Multi-Agent Systems*, Utrecht (The Netherlands).

[40] Moszkowski, B. (1996) The programming language Tempura. *J. Symb. Comput.*, **22**, 730–733.

[41] Brisset, P. (2000) A Case Study in Java Software Verification. *Appeared in Workshop on Security, Middleware, and Languages*, Stockholm.

[42] Sekar, R., Venkatakrishnan, V.N., Basu, S., Bhatkar, S. and Du Varney, D. (2003) Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. *ACM Symp. Operating Systems Principles*, Bolton Landing (USA).

[43] Damianou, N., Dulay, N., Lupu, E.C. and Sloman, M.S. (2001) The Ponder Policy Specification Language. *Presented at Policy, in Workshop on Policies for Distributed Systems and Networks*, Bristol (UK).

[44] Capra, L., Emmerich, W. and Mascolo, C. (2001) Reflective Middleware Solutions for Context-Aware Applications. In Yonezawa, A. and Matsuoka, S. (eds) *Proc. Reflection. The 3rd Int. Conf. on Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto (Japan). LNCS 2192. Springer.

[45] Capra, L., Emmerich, W. and Mascolo, C. (2003) CARISMA: context aware reflective middleware system for mobile applications. *IEEE Trans. Softw. Eng.*, **29**, 929–945.

[46] Mascolo, C., Capra, L., Zachariadis, S. and Emmerich, W. (2002) XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Int. J. Wirel. Pers. Commun.*, **21**, pp. 77–103. Kluwer Academic Publisher.

[47] Brörkens, M. and Möller, M. (2002) Jassda Trace Assertions, Runtime Checking the Dynamic of Java Programs. In Schieferdecker, I., König, H. and Wolisz, A. (eds) *Trends in Testing Communicating Systems*, pp. 39–48. Berlin, Germany.

[48] Emmerich, W. (2000) Software Engineering and Middleware. A Roadmap. *The Future of Software Engineering—22nd Int. Conf. on Software Engineering*, Limerick (Ireland). pp. 117–129. ACM Press.

[49] Goldberg, A. and Havelund, K. (2003) Instrumentation of Java Bytecode for Runtime Analysis. *Formal Techniques for Java-like Programs*. Technical Reports from ETH Zurich 408. ETH Zurich, Switzerland.

[50] Cohen, G., Chase, J. and Kaminsky, D. (1998). Automatic Program Transformation with JOIE. *Proc. USENIX Annual Technical Symp*, New Orleans (USA).

[51] Necula, G. and Lee, P. (1996) Proof-Carrying Code. Technical Report CMU-CS-96-165. Carnegie Mellon University, Noviembre.

[52] Necula, G. and Lee P. (1988) The Design and Implementation of a Certifying Compiler. *Proc. 1988 ACM SIGPLAN Conf. on Programming Language Design and Impl*, Atlanta (USA).

[53] Meyer, B. (2000) *Object-Oriented Software Construction* (2nd edn). Prentice Hall, Upper Saddle River, NJ.