City Research Online

City, University of London Institutional Repository

Citation: Kloukinas, C. & Spanoudakis, G. (2007). A pattern-driven framework for monitoring security and dependability. Lecture Notes in Computer Science: Trust, Privacy and Security in Digital Business, 4657, pp. 210-218. doi: 10.1007/978-3-540-74409-2_23

This is the unspecified version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/2892/

Link to published version: https://doi.org/10.1007/978-3-540-74409-2_23

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

A Pattern-Driven Framework for Monitoring Security and Dependability

Christos Kloukinas and George Spanoudakis

Department of Computing, The City University, London, EC1V 0HB, U.K.
Email: {C.Kloukinas,G.Spanoudakis}@soi.city.ac.uk

Abstract. In this paper we describe a framework that supports the dynamic configuration, adaptation and monitoring of systems that need to guarantee specific security and dependability (S&D) properties whilst operating in distributed settings. The framework is based on *patterns* providing abstract specifications of implementation solutions that can be used by systems in order to achieve specific S&D properties. The focus herein will be on the monitoring aspects of the framework which allow it to adapt to violations of the S&D requirements and changes to the current context.

1 Introduction

Ensuring security and dependability in systems which operate in highly distributed environments and frequently changing contexts (e.g. changing networks and system deployment infrastructures), whilst maintaining system interoperability and adaptability, is one of the major challenges of current research in the area of security and dependability [1], where systems need to adapt to dynamic changes in their context. This necessitates the incorporation of mechanisms that can monitor a system's operation and report violations of S&D requirements that would require the deployment of alternative S&D mechanisms.

In this paper, we present a framework that is being developed as part of the European research project SERENITY [1] to address the above challenges. This framework is driven by S&D patterns which specify reusable architectural solutions for S&D requirements, the contextual conditions under which these solutions are applicable, and rules that need to be monitored at run-time to ensure that the implementation of the pattern behaves correctly. The framework is responsible for selecting the patterns which are appropriate for fulfilling the S&D requirements of a system in specific operational contexts, as well as, activating and integrating the implementations of these patterns with the system at runtime. The

[1] http://www.serenity-project.org/motivations-&-objectives.php

framework can also monitor the execution of the system and the implementations of the S&D patterns, and take corrective actions if a violation of rules or contextual conditions of the patterns is identified.

The general architecture and functions of this framework have been introduced in [2]. Our focus in this paper is to describe the support that the framework provides for *system monitoring* at run time and present the use of the S&D patterns in monitoring and the mechanisms that the framework incorporates to support this activity. The rest of this paper is structured as follows. In section 2, we present an example of a system which will be used throughout the paper to illustrate the operations of the framework. In section 3, we present the general architecture of the framework and discuss the S&D patterns and other artefacts which are deployed during monitoring. In section 4, we discuss the monitoring life cycle that is realised by the framework and how it is driven by the dynamic selection, activation and deactivation of S&D patterns. In section 5, we overview related work and, finally, in section 6 we give some concluding remarks and outline plans for future work.

2 Motivating Example

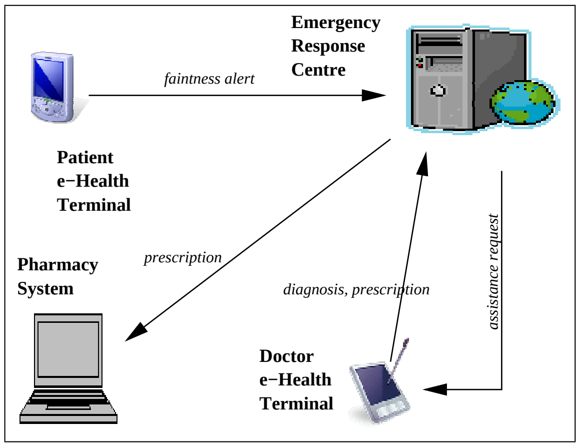


Fig. 1. e-Healthcare system

The system that we use to illustrate the function of the S&D framework that we describe in this paper is an *e-healthcare system* whose objective is to support the monitoring, assistance, and provision of medication to patients who have been discharged from hospitals with critical medical conditions [3]. In an operational scenario of this system, a patient does not feel well and sends through his *patient e-health terminal (PHT)* a request for assistance to the *emergency response centre (ERC)*. To establish the cause of the problem, ERC retrieves the patient's medical record from its internal database. From this record, ERC establishes that the patient's doctor is on vacation and contacts an alternative doctor D whose expertise matches with the expertise of the patient's doctor. Doctor D receives this message on her *doctor e-health terminal (DHT)* and replies immediately. ERC verifies D's identity and sends the patient's medical data to DHT. D creates an electronic prescription on her DHT, sends it to ERC, which subsequently forwards it to the *pharmacy system (PhS)* that is closest to the patient's location. The pharmacy delivers the medicines to the patient, and PhS confirms the dispatch to the ERC.

3 Overview of the Framework

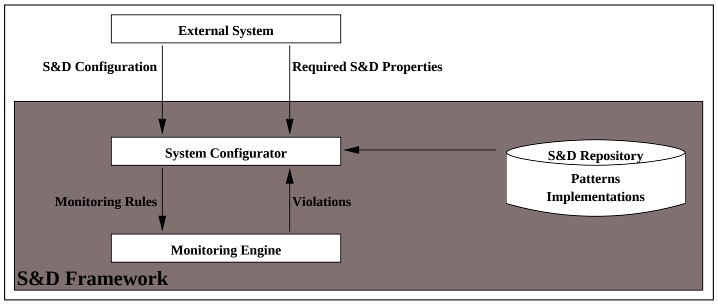


Fig. 2. Architecture of the S&D framework

The generic architecture of the S&D framework is shown in Fig. 2. The *S&D configurator* accepts as input the S&D configuration of an external system and the S&D properties that this system wants to realise. Then the S&D configurator selects an S&D pattern which can provide the required S&D properties and also selects a concrete implementation of this S&D pattern which is applicable in the particular setting. Then the S&D configurator sends the rules that need to be monitored for the specific pattern and implementation to the monitoring engine, and activates the implementation.

The *monitoring engine (ME)* gets the rules that should be monitored and starts the monitoring activity. During this activity, the engine gets events concerning the state of the external system and the selected implementation from *event captors* and sends notifications of violations of monitoring rules to the S&D configurator. *S&D implementations* include operational runtime components that can be used to realise the S&D properties of the pattern that they are associated with. They also include event captors which provide the events required for checking the monitoring rules of the S&D patterns. S&D implementations are activated and deactivated dynamically by the S&D configurator through different mechanisms depending on their type.

3.1 The Basic Artefacts

Requirements and Properties S&D requirements of systems are expressed as S&D properties which need to hold. More specifically, a system provides the framework with a configuration file specifying: (i) the required S&D properties, (ii) the part of the system's architecture that each property relates to, and (iii) the *attack/fault model* (*afm* - itself expressed as an S&D property) under which the property should be guaranteed, in an assume-guarantee type of reasoning: $afm_a \Rightarrow prop_b$. Using (i)-(iii), the S&D framework can select an appropriate pattern for the relevant property.

The properties and attack/fault models are represented abstractly as keywords and their interdependencies as implications, e.g., $prop_a \Rightarrow prop_b$. By doing so, it is easy to dynamically check whether the assumed attack/fault model is more constrained than that of a pattern, i.e., $afm_{sys} \Rightarrow afm_{pat}$, and whether the property required by the system is weaker than the property provided by the pattern, i.e., $prop_{pat} \Rightarrow prop_{sys}$.

Patterns A simplified [2] example of an S&D pattern (I&C) is shown in Fig. 3. I&C provides two *properties*, integrity and confidentiality, under any attack model. It contains *monitoring rules* for verifying the properties at runtime, *assumptions* which provide extra information about the system behaviour, and *contextual conditions* under which the pattern is applicable. Relation $RequiresRule_{pat}$ helps determine the subset of rules which should be monitored, to avoid wasting resources if we do not need all properties, while relation $DependsOn_{pat}$ indicates which assumptions should be used when particular rules need to be monitored. Finally, the pattern contains an *architectural description* of the offered solution, which

[2] More details about the contents of S&D patterns and the scheme for describing them can be found in [4].

describes its *components*, i.e., what the pattern provides for realising the solution, the *parameters*, i.e., partially unknown components of the system which will use the pattern, and the *connectors* which link these together (shown as arrows).

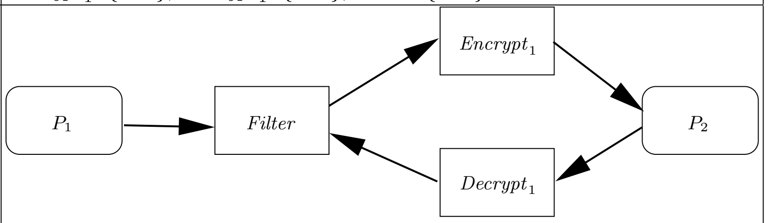
Properties	True $\Rightarrow$ Confidentiality, True $\Rightarrow$ Integrity
Parameters	P_1: $\{\ldots\}$, P_2: $\{\ldots\}$
Components	$\mathit{Encrypt}_1$: $\{\ldots\}$, $\mathit{Decrypt}_1$: $\{\ldots\}$, Filter: $\{\ldots\}$
Architectural Description	 P_1 Filter $\mathit{Encrypt}_1$ $\mathit{Decrypt}_1$ P_2
Rule$_1$	Happens $(e(id, \mathit{Filter}, P_1, \mathit{RES}, \_X, \mathit{Filter}), t_1, \Re(t_1, t_1)) \Rightarrow$ Happens $(e(id, P_1, \mathit{Filter}, \mathit{REQ}, \_X, \mathit{Filter}), t_2, \Re(t_2, t_1))$
Rule$_2$	Happens $(e(id, \mathit{Filter}, \mathit{Encrypt}_1, \mathit{REQ}, \_X, \mathit{Filter}), t_1, \Re(t_1, t_1)) \Rightarrow$ Happens $(e(id, \mathit{Encrypt}_1, \mathit{Filter}, \mathit{RES}, \_X, \mathit{Filter}), t_2, \Re(t_1, t_1 + T))$
Rule$_3$	Happens $(e(id, \mathit{Encrypt}_1, P_2, \mathit{REQ}, \_X, \mathit{Encrypt}_1), t_1, \Re(t_1, t_1)) \Rightarrow$ HoldsAt $(\mathit{authorised}(P_2, P_1), t_1)$
Assumption$_1$	Happens $(e(id, P_1, \_X, \mathit{RES}, \mathit{authorise}(P_2, \mathit{result}), P_1), t_1, \Re(t_1, t_1))$ $\wedge\ \mathit{result} = \text{True} \Rightarrow$ Initiates $(e(id, P_1, \_X, \mathit{RES}, \mathit{authorise}(P_2, \mathit{result}), P_1), \mathit{authorised}(P_2, P_1), t_1)$
Context Condition$_1$ (CC$_1$)	Happens $(e(id1, \_Y, \_Z, \mathit{REQ} \mid \mathit{RES}, \_X, \mathit{Filter}), t_1, \Re(t_1, t_1))$ $\wedge\ (\_Y = \mathit{Filter} \vee \_Z = \mathit{Filter}) \Rightarrow$ Happens (exec :$e(id2, \text{ME}, \mathit{Filter}, \mathit{REQ}, \text{getCertificate}(), \text{ME}), t_2, \Re(t_1, t_1 + 1))$ $\wedge$Happens $(e(id3, \mathit{Filter}, \text{ME}, \mathit{RES}, \text{getCertificate}(\mathit{cert}), \text{ME}), t_3, \Re(t_2, t_2 + T))$ $\wedge$ valid(cert) = True
$\mathit{RequiresRule}_{pat}$	$\{$(Integrity, Rule$_1$), (Confidentiality, Rule$_2$), (Confidentiality, Rule$_3$), (Integrity, CC$_1$)$\}$
$\mathit{DependsOn}_{pat}$	$\{$(Rule$_3$, Assumption$_1$)$\}$

Fig. 3. A simplified pattern example of integrity and confidentiality

Rules, context conditions and assumptions are specified in Event Calculus (EC) [5]. An event $e(\mathit{ID}, \mathit{sender}, \mathit{receiver}, \mathit{status}, \mathit{operation}, \mathit{source})$, provides us with its *source*, that is the component from which the occurrence of the *operation* has been captured (may be different from either *sender* or *receiver*), and their *status*, that is whether the *operation* is a request (REQ) or a response (RES). Fluents are represented as relations between objects of the general form: $f(o_1, \cdots, o_n)$.

Rule$_1$ in the I&C pattern describes an integrity constraint, where for each response to an operation call that P_1 receives from *Filter*, there should be a matching earlier call of this operation that was sent from P_1 to *Filter*. Rule$_2$ checks the (bounded) availability of $\mathit{Encrypt}_1$, by asking that *Filter* should respond to an operation $\_X$ within T time units. Rule$_3$ checks if the recipient P_2 of any message $\_X$ from $Encrypt_1$ is authorised by P_1 to receive messages at the time of dispatch of $\_X$. Finally, the context condition (CC$_1$) examines the validity of the certificate of the pattern every time that an operation is called on/by *Filter*. If the certificate has been revoked between any of these points, then the pattern is no longer applicable and must be deactivated.

4 The Monitoring Lifecycle

The typical operational scenario of the S&D framework involves: (i) the selection of a pattern that can provide the properties required by a system, (ii) the activation of an appropriate implementation for it and the monitoring of the pattern rules, and, (iii) the deactivation of the pattern if it is no longer relevant to the external system of concern or cannot be applied in the current context. In the following, we describe how the S&D framework performs these activities.

Selection of Patterns Based on the system S&D configuration file, the S&D framework searches its pattern repository, to identify patterns which offer the required properties ($RProp_j$), given the specific attack/fault models (AFM_i). More specifically, it computes the $TolerableAttacks = \{afm : AFM_i \Rightarrow afm\}$ and $ProvidedProperties = \{prop : prop \Rightarrow RProp_j\}$ and uses these to find the *CandidatePatterns*, which provide the property $afm \Rightarrow prop$. Then the framework finds the *RealisableCandidates* which have currently applicable implementations. At this stage, extra constraints specified by the system configuration are used to sort the set of realisable candidate patterns with respect to how closely they match the user's criteria, e.g. the maximum cost of the provided implementation, the identity of its provider, etc. Then, the closest match is considered for the most difficult part of the search, i.e., selecting a pattern which is *architecturally compatible* with the system. The problem of architectural compatibility is ensuring that the system components which require a property will be correctly assigned to the parameters of the pattern. This architectural match is performed through architectural unification [6]. The selection process ends when the S&D framework has found an architecturally compatible pattern in the ordered set of *RealisableCandidates*. In reference to the example of Fig. 1, we will assume that the configurator has selected the pattern of Fig. 3 as a realisable candidate pattern, using the substitutions $\{P_1 \rightarrow ERC, P_2 \rightarrow DHT, \_X \rightarrow assist(\cdots)\}$, where $assist(\cdots)$ is the operation that the ERC is calling on the DHT.

Activation of Patterns The activation of patterns by the S&D framework has two major steps with respect to monitoring: (1) the activation of monitoring rules by the monitoring engine, and (2) the attachment of the event collectors to the system/pattern components in order to generate the events required for monitoring.

The activation of monitoring rules happens according to the following steps, using the information that the I&C pattern has been selected for both its properties $\mathit{SelectedFor}_{pat} = \{(\mathit{Confidentiality}, \mathit{Integrity})\}$:

Computations	Results
$\mathit{InitRules}_{pat} = \mathit{CC}_{pat} \cup \{r : \exists \mathit{prop} \in \mathit{SelectedFor}_{pat} \mid (\mathit{prop}, r) \in \mathit{RequiresRule}_{pat}\}$	$\mathit{InitRules}_{pat} = \{\mathrm{CC}_1, \mathrm{Rule}_1, \mathrm{Rule}_2, \mathrm{Rule}_3\}$
$\mathit{FinalRules}_{pat} = \mathit{InitRules}_{pat} \cup \bigcup_{r \in \mathit{InitRules}_{pat}} \mathit{DependsOn}_{pat}(r)$	$\{\mathrm{CC}_1, \mathrm{Rule}_1, \mathrm{Rule}_2, \mathrm{Rule}_3, \mathrm{Assumption}_1\}$
$\mathit{ActiveRules}_{pat} = \mathrm{substitute}(\mathit{FinalRules}_{pat}, \mathit{substitutionlist})$	$\mathit{substitutionlist} = (P_1 \to \mathit{ERC}, P_2 \to \mathit{DHT}, \_X \to \mathit{assist}(\cdots)))$

Once the monitoring rules of the selected pattern have been instantiated and activated, the event collectors of the respective S&D implementation are activated. This process uses $\mathit{ActiveRules}_{pat}$:

Computations	Results/Comments
$\mathit{EventsOfInterest}_{pat} = \bigcup_{r \in \mathit{ActiveRules}_{pat}} \mathit{Contains}_{pat}(r)$	$\mathit{EventsOfInterest}_{pat} = \{ \mathit{ev}(\mathit{id}, \mathit{ERC}, \mathit{Filter}, \mathit{REQ}, \mathit{assist}(\cdots), \mathit{Filter}), \mathit{ev}(\mathit{id}, \mathit{Filter}, \mathit{ERC}, \mathit{RES}, \mathit{assist}(\cdots), \mathit{Filter}), \cdots\}$
$\mathit{SourceOf}_{pat}(e) = c$	*Filter* (for all events)
Find the event collectors for each event e: $\mathit{CollectedBy}_{imp}(\mathit{SourceOf}_{pat}(e), e)$	*From the the configuration of the selected S&D implementation*

The monitoring engine checks the activated monitoring rules as described in [7]. If a rule is violated, the engine logs the violation and performs the control action which was specified in the system configuration, if any, to notify the system. If the violated rule is part of the pattern's context conditions, then the framework configurator is notified in order to deactivate the pattern and replace it with a new one.

Deactivation of patterns When a context condition is violated or when the S&D requirements change, e.g., due to legal reasons, then the pattern needs to be deactivated and replaced by another. Replacing a pattern entails the deactivation of the monitoring rules and assumptions, the detachment of the event collectors which collect the events for these rules and the deactivation of its implementation. Even though the $\mathit{ActiveRules}_{pat}$ are easy to deactivate, event collectors should only be deactivated if they are not also being used by other implementations. Therefore, the S&D

configurator needs to identify the collectors which are used exclusively by the current pattern and deactivated these only.

5 Related Work

The objective of the framework that we present in this paper is two-fold: (a) to provide runtime support to external systems for the realisation of specific S&D properties, presented in more detail in [2], and (b) to monitor the effectiveness and adequacy of the support that it provides in specific operational contexts. The approach that we advocate for (b) is related to *security monitoring systems*, which can be distinguished into *firewalls* and *intrusion detection systems* [8,9], *intrusion prevention systems* [10,11], and *access control systems* [12,13]. Firewalls control access on packets entering or leaving local networks to protect them from external networks, thus do not consider the application layer and cannot protect against internal threats or monitor general security properties. Intrusion detection systems also aim to detect attacks at the network layer based on models of expected user/system behaviour but do not always have the control capability to prevent attacks. A combination of attack detection and prevention capabilities is provided by intrusion prevention systems. Access control systems aim to restrict access to sensitive information based on pre-assigned rights for accessing specific information objects to different subjects (e.g. system component), the requester's role in an organisation (*role based access control systems*), or access policies combining credentials of users with the context of the system (*context based access control*). Such systems can monitor information access but not other, more general, properties which are supported by our approach. Furthermore, they cannot adapt and integrate complex security solutions to running systems [1].

Our approach also relates to *general purpose runtime monitoring systems*, which focus on the verification of program behaviour against properties specified at some temporal logic or on requirements monitoring, e.g. [14,15]. Many of the former systems focus on runtime verification of Java code [16,17] where events record changes of internal program variable values and/or invocations and returns of program methods. The latter systems express requirements in some high level formal specification language and subsequently assume the refinement and mapping of these requirements onto patterns of events whose occurrence would indicate their violation at run-time. This transformation is the responsibility of system providers, e.g. [14].

The framework that we present in this paper can support the monitoring of general properties for software systems including security properties [18]. Its main difference from existing work is that monitoring is driven by S&D patterns which define the rules that should be monitored at different stages and contexts of a system's operation, in order to ensure specific security properties. Furthermore, the generation of events in this framework is performed by pattern implementations and thus there is no need for explicit code instrumentation or developing other types of event emission methods.

6 Conclusions

In this paper, we described the monitoring-related aspects of a framework [2] that supports the dynamic configuration, adaptation and monitoring of systems that need to guarantee specific security and dependability properties whilst operating in distributed settings. The framework is based on *patterns* [4] providing specifications of implementation solutions that can be used by systems in order to achieve specific security and dependability properties. Patterns identify contextual conditions which need to hold in order to guarantee the effectiveness of the solutions that they describe, and rules that should be monitored at runtime to check that these conditions are satisfied and the offered solutions do indeed comply with the required security and dependability properties.

Based on the security and the dependability properties which are required by external systems, the framework can automatically select patterns and concrete implementations, integrate them with the system, and monitor the behaviour of the integrated entity to check the effectiveness of the adopted solutions in it. The framework can also take certain control actions when there are runtime violations of the monitored rules. These actions may include the selection and activation of other patterns if the current ones fail to meet the requirements, the activation of additional monitoring activities, and the suspension of the system's operation.

Currently, we are working on the introduction of mechanisms for detecting potential threats to S&D requirements and the provision of detailed diagnostic information for the detected violations of the S&D pattern rules and contextual conditions. We are also looking onto mechanisms for the effective distribution of rules onto different monitors in order to optimise the monitoring performance of the framework.

References

1. Maña, A., et al.: Security engineering for ambient intelligence: A manifesto. In: Integrating Security and Software Engineering: Advances and Future Vision. Idea Group Publishing (2006) 244–270
2. Sanchez-Cid, F., et al.: Software engineering techniques applied to AmI: Security patterns. In: Developing Ambient Intelligence: Proc. of the First Int. Conf. on Ambient Intelligence Developments (AmID'06), Sophia-Antipolis, France, Springer (2006)
3. Campadello, S., et al.: S&D requirements specification. Deliverable A7.D2.1, SERENITY Project (2006) Available from `http://www.serenity-forum.org`.
4. Maña, A., et al.: Patterns and integration schemes languages. Deliverable A5.D2.1, SERENITY Project (2006) Available from `http://www.serenity-forum.org`.
5. Shanahan, M.P.: The event calculus explained. In: Artificial Intelligence Today. Volume 1600 of Lecture Notes in Artificial Intelligence. (1999) 409–430
6. Melton, R., Garlan, D.: Architectural Unification. In: Proceedings of CASCON'97, Ontario, Canada (1997)
7. Spanoudakis, G., Mahbub, K.: Non intrusive monitoring of service based systems. International Journal of Cooperative Information Systems **15** (2006) 325–358
8. Axelsson, S.: Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Dept. of Computer Engineering, Chalmers Univ. (2000)
9. Hofmeyr, S.A., Forrest, S.: Architecture for an artificial immune system. Evolutionary Computation **7** (2000) 1289–1296
10. Anagnostakis, K., et al.: Detecting targeted attacks using shadow honeypots. In: Proc. of the 14th USENIX Security Symposium. (2005)
11. Labbe, K., et al.: A methodology for evaluation of host-based intrusion prevention systems and its application. In: Proc. of the 7th IEEE Work. on Information Assurance. (2006)
12. Corradi, A., et al.: Context-based access control management in ubiquitous environments. In: Third IEEE Int. Symp. on Network Computing and Applications. (2004) 253–260
13. Hulsebosch, J., et al.: Context sensitive access control. In: Proc. of the Tenth ACM Symp. on Access Control Models and Technologies, SACMAT'05. (2005) 111–119
14. Robinson, W.: Monitoring software requirements using instrumented code. In: Proc. of the Hawaii Int. Conf. on Systems Sciences, Hawaii, USA (2002)
15. Feather, M., et al.: Reconciling system requirements and runtime behaviour. In: Proc. of 9th Int. Work. on Software Specification & Design. (1998)
16. Kannan, S., et al.: Runtime monitoring and steering based on formal specifications. In: Workshop on Modeling Software System Structures in a Fastly Moving Scenario. (2000)
17. Kim, M., et al.: Java-MaC: a runtime assurance tool for Java programs. Electr. Notes in Theoretical Computer Science **55** (2001)
18. Spanoudakis, G., Kloukinas, C., Androutsopoulos, K.: Towards security monitoring patterns. In: ACM Symposium on Applied Computing (SAC07) - Track on Software Verification. Volume 2., Seoul, Korea, ACM (2007) 1518–1525