Extending LSCs for Behavioral Signature Modeling

Sven Patzina, Lars Patzina, Andy Schürr

Extending LSCs for Behavioral Signature Modeling

Sven Patzina[1], Lars Patzina[2], and Andy Schürr[1]

[1] Real-time Systems Lab, TU Darmstadt, Darmstadt, Germany
{sven.patzina, andy.schuerr}@es.tu-darmstadt.de
[2] Center for Advanced Security Research Darmstadt (CASED), Germany
lars.patzina@cased.de

Abstract. Driven by technical innovation, embedded systems are becoming increasingly interconnected and have to be secured against failures and threats from the outside world. For this purpose, we have defined an integrated model-based development process for security monitors which requires an expressive, formally well-defined, and easy to learn behavioral signature language. In this paper, we demonstrate that Live Sequence Charts (LSCs) are adequate for the specification of behavioral signatures. To satisfy all requirements and enable compact modeling, we extend LSCs by concepts that fit well to the spirit of LSCs.

1 Introduction

Driven by technical innovation, embedded systems are becoming increasingly interconnected. Thus, they cannot be considered as being separated from the outside world, even though many of them were developed as such. Often, little attention has been paid to security mechanisms, such as encryption and safe component design for defense against attacks. Groll and Ruland [3] show such weaknesses for passive and active attacks in modern networks in the automotive domain and postulate that additional security measures are needed. Furthermore, Koscher et al. [7] identify the CAN bus protocol as a major security drawback in modern automobiles. For subsequent protection of these systems Papadimitratos et al. [12] propose secured communication in the car and the development of a secure architecture to improve privacy and security.

Even when all these proposed techniques are applied during the development of an embedded system, in the majority of cases it is impossible to eliminate all security vulnerabilities and to foresee all possible attacks. Considering huge heterogeneous systems or components, it is often economically or technically infeasible to secure them against external adversaries retroactively. Therefore, systems cannot be considered as safe, either due to unknown vulnerabilities, or due to the required integration of legacy components.

To secure such systems, Kumar [8] proposes monitoring the system at runtime, which permits the detection of attacks that exploit previously unknown errors and security vulnerabilities. The two most common approaches for this

purpose are, firstly, signature-based detection, which uses predefined attack descriptions and, secondly, anomaly detection, which recognizes the faulty behavior of a system by detecting deviations from the intended behavior. However, signature-based detection is only able to detect attacks that are similar to known vulnerabilities and attack classes, which leads to a low false-positive rate, but also to an insufficient number of matches. In contrast, anomaly detection is able to reveal unknown attacks by observing their impact on the system, but suffers from a high false-positive rate. These false matches have to be handled by user interaction to evaluate the threat or by self-healing techniques to transfer the system to a secure and stable state.

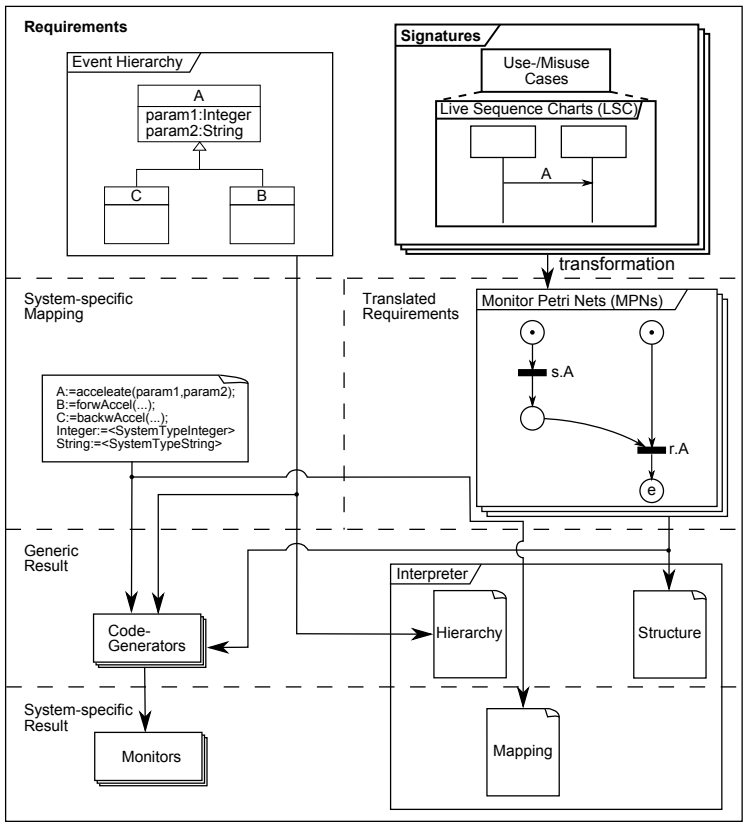


Fig. 1. Model-based security engineering process.

Our goal is a comprehensible, model-based development process, based on the Model Driven Architecture concept, to automatically generate security monitors from a specification consisting of Live Sequence Charts (LSCs) structured by use and misuse cases. The process – depicted in Fig. 1 – starts in the requirements

phase, where the intended system behavior is modeled as use cases and known attack patterns and attack classes are modeled as misuse cases [15]. These abstract specifications are described in more detail with LSCs. Through automatic transformation and weaving of system-specific information, a security monitor is automatically generated, using the intermediate Monitor Petri net language [13].

Our contributions in this paper are:

- the evaluation of LSCs as a signature language.
- the proposal of extensions to classic LSCs, to overcome their limitations.
- the application of extended LSCs to an example scenario.

In this paper, we show that LSCs are applicable to the description of behavioral signatures. In Sect. 2, various approaches for modeling policies, together with their advantages and disadvantages, are presented. Subsequently, the requirements that have to be satisfied by a behavior specification language are introduced in Sect. 3. According to these, we present in Sect. 4 the required conditions and extensions to satisfy the requirements for a policy language. Section 5 draws a conclusion and describes starting points for future work.

2 Related Work

The development process we present for security monitors requires a language to specify intended and forbidden behavior as signatures. This language must be able to describe functional and non-functional requirements (NFRs) that need to be monitored. Specifying NFRs is more challenging than specifying functional requirements. These NFRs are often described in a natural language and are therefore very abstract. To obtain a more formal description of these policies and behavioral signatures, special policy languages or temporal logic are commonly used for Intrusion Detection Systems (IDSs).

One of these concepts are expert systems (ES) such as CLIPS [2] and P-BEST [9]. These systems are based on inference rules with an if-then structure. If the guard of a rule is satisfied, then the specified action is performed on facts that describe the monitored system. Thus, the basis of facts is modified by the system and by the rules, which can trigger new actions. Furthermore, temporal rules have to be formulated over the facts, which is not at all intuitive. Another, better fitting, approach to formulate temporal aspects is the usage of temporal logic as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL).

These languages are often hard to learn and are not comprehensible for non-practitioners. To overcome these shortcomings, several extensions to the de facto standard of the OMG – the UML – have been proposed. A lightweight extension of the UML, called UMLsec [6], extends the system specification by security properties, such as encryption of communication. Therefore, it provides a UML profile, defining stereotypes and tagged values, to annotate the system model. In contrast, SecureUML [10] is a heavyweight extension of the UML that changes the meta model. This dialect of the UML is designed for modeling role-based

access control restrictions. Both were developed for extending UML to describe non-functional security constraints in a model-based development process. However, they lack the ability to describe behavioral signatures needed for an IDS. To close this gap, Hussein et al. [5] propose a UML profile providing stereotypes to annotate several UML diagrams. They employ use cases for the scenario description, classes for the structure of the system, and state machines to model potential steps of the behavior of the system.

Beside UML statecharts, other state/transition-oriented languages are used to model behavioral signatures. A variant with a simple structure is used by STAT [19], where states are specified by invariants, and actions and transitions are annotated by conditions, events, and actions. By omitting fork and join, a transition can only have one predecessor and one successor, to ease the interpretation. This results in an explosion of states when modeling concurrent behavior, because every permutation has to be modeled as a separate state. In contrast to STAT, IDIOT [8] uses expressive Coloured Petri nets, annotated with the general purpose functional programming language ML. This complex syntax, consisting of Petri nets and a functional programming language, leads to powerful, but hard to understand, specifications.

To ease the modeling and allow non-practitioners to understand the specification, a more high-level approach for signature modeling is desirable. So Massacci and Naliuka [11] use UML sequence diagrams (SDs) extended with linear temporal logic for modeling behavioral signatures. However, the semantics of SDs is not suited to model deontic constraints (obligation, permission and prohibition). Therefore, they use them in a not UML fashioned way, which conflicts with the use of the standard and hampers the developer in using existing UML knowledge. With the same drawbacks, Solhaug et al.[18] pursue a very similar approach, by using SDs to define policy specifications. However, they use the STAIRS semantics to interpret their SDs and state that Live Sequence Charts "could serve as an alternative for interpreting policy rules".

If LSCs[4] can be used to describe the semantics of policies, why not use them as a specification in their original semantics? In the following, we will examine what the requirements for our monitor development process are and describe how these requirements can be satisfied by LSCs.

3 Requirements

To be able to define and evaluate a signature language, it is necessary to identify its requirements that have to be met. As discussed in Sect. 2, there are different possibilities to model attacks and intended behavior. Since we use the specification language in a model-based development process that should be also understandable to non-practitioners, it has to possess a graphical syntax. Though, Live Sequence Charts satisfy this high-level requirement, we take a closer look on important requirements for a software engineering process and especially for a policy (signature) language.

In this context Smith et al. [17] propose several criteria for security and software engineering processes. Adopted to our monitor development process these are:

1 - Easy to learn The access to the modeling language should be easy. So a flat learning curve for software developers is desirable.
2 - Comprehensible A fundamental comprehension of the modeled systems should be possible for non-practitioners.
3 - Predictive The modeled specifications should have a defined semantics to be able to analyze or simulate them in order to reveal non-obvious properties.
4 - Effective transition to implementation The transitions between models up to the code generation have to be properly defined and implemented.
5 - Cost-effective Building the model has to be less expensive than other appropriate ways of building the system.
6 - Expressive The language has to be expressive enough to represent all kinds of key concepts without losing important details.

(1) Most of the diagram types used for our specification are known by software engineers because they are based on diagrams borrowed from the UML 2.0. So UML use cases are extended by the concept of misuse cases, which allow the modeling of unintended behavior and attacks. Class diagrams are used to build a hierarchy of messages and describe the structure of the system. Live Sequence Charts are an extension of Message Sequence Charts and UML 2 Sequence diagrams and preserve the graphical appeal and intuitiveness of MSCs [20]. This enables the developer to start modeling right away by extending his knowledge stepwise for the special properties of the languages.

(2) In contrast to many existing special policy languages like expert systems or state/transition-based approaches, where the human readability is no core issue [16], the LSCs can be easier understood by non-practitioners.

(3) LSCs are well defined and posses a strict formal semantics [1]. These descriptions are translated in the formally defined Monitor Petri nets [13] to enable a generic code generation for different target platforms. For this reason, simulation and analysis can be performed to verify the correctness of the specification.

(4) In our development process, we use the concept of the Model Driven Development proposed by the OMG[3] for the UML. The transformations are defined by a graphical graph transformation language (Story Driven Modeling) working on repositories generated by MOFLON[4].

(5) Costs can be reduced, because the system developers do not have to learn a special unintuitive language and can use specifications from the early requirements phase through the whole development process by refining them.

(6) In our case, it is important that we support all necessary constructs for behavioral signature modeling. Therefore, Schmerl [14] has evaluated several policy languages and proposed requirements that a general policy language based on Petri nets has to fulfill. He has categorized event patterns with respect to

[3] Object Management Group: www.omg.org
[4] MOFLON meta-CASE tool: www.moflon.org

several aspects of the semantics model listed in Table 1. In the next section, we show how these requirements are met by LSCs.

Beside these requirements, the modeling of obligations, permissions, and prohibitions have to be supported. As Soulhaug et al. [18] have stated, these deontic modalities can be expressed by LSC by nature. So obligations can be expressed by use cases described by existential charts, permissions by use cases with universal charts (composed of pre- and mainchart), and prohibitions by misuse cases with universal charts.

4 LSCs as Behavioral Signature Modelling Language

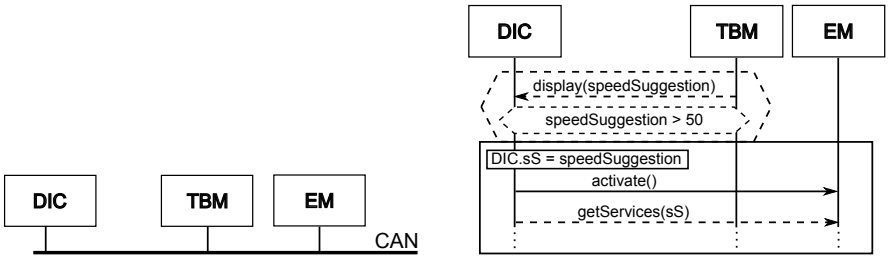
In the requirements phase of our development process, Live Sequence Charts are used in combination with structural specifications. Therefore, UML class diagrams describe the structure of the system and specify the participants and their relations. To model the relation between single signatures, UML use cases extended with the concept of misuse cases are used. They declare whether the signature describes an intended behavior (use case) or a faulty behavior or attack (misuse case).

As a detailed specification language for modeling use and misuse cases, we exploit the expressiveness of Live Sequence Charts. After pointing out the requirements that are crucial for a policy language in Sect. 3, we demonstrate by example how LSCs can satisfy these. In the following, we use a scenario, shown in Fig. 2a), based on a CAN bus. Koscher et al. [7] have shown that there are security threats in modern automobiles. Many of them result from the weaknesses of the CAN bus protocol, because packets of this protocol do not include authenticator fields or identifiers for the source. The CAN ID header only contains information about the packet type. Additionally, these packets are broadcast to every node in the network that decides by itself if the packet is relevant. So one compromised component is enough to inject messages on the CAN bus and, thereby, control other nodes of the network. In this way, false information could be displayed on the Driver Information Center (DIC) that is connected to the bus. These corrupted messages are injected by two control units communicating with the outside world. One is a wireless communication module for toll collection (TBM) and the other communicates with some enterprise roadside units (EM).

Before discussing how the requirements raised in Sect. 3 are complied by LSCs, we explain the basic concepts of LSCs by the example in Fig. 2b. In this signature, three instances are involved called *DIC*, *TBM* and *EM*. The vertical lines are lifelines defining a partial order in time from the top to the bottom. The dashed hexagon, a prechart, is special to LSC *universal charts* and represents a precondition that has to be fulfilled before the main chart, the rectangle below, is valid. As in message sequence charts, function calls or messages are modeled by arrows between the lifelines of the instances, conditions as hexagons and assignments of values in rectangles. Thereby, all dashed lines represent *cold* (optional) and solid lines *hot* (mandatory) elements. The prechart

Type of sequences	
Sequence	Several events with a strict sequential order.
Conjunction	Several event patterns that can occur in arbitrary order.
Negation	An event pattern that must not occur.
Disjunction	One event pattern of several is possible.
Simultaneous	Two events that occur at the same time.
Type of Iterations	
Exact	A pattern has to occur n times.
At least	A pattern has to occur at least n times.
At most	A pattern has to occur at most n times.
Continuity	
Continuous	Every event that occurs has to be modeled in the signature.
Non-continuous	The signature is matched if all modeled events have occurred. All additional events between are ignored.
Concurrency	
Non-overlapping	Two or more sequences of events have to occur sequentially. They must not share events during matching.
Overlapping	Two or more sequences of events are allowed to share events during matching .
Context conditions	
Intra-step conditions	A simple boolean expression on an event occurrence.
Inter-step conditions	A complex condition between properties of several events.
Matching rules	
First	The first event matching is bounded.
Last	The last event of several occurrences is bounded.
All	All matching events are bounded.
Type of consumption	
Consuming	In a signature an occurred event is only used for one match in the modeled pattern.
Non-Consuming	In a signature an occurred event is used for several matches in one pattern.

Table 1. Requirements for a behavioral signature language.



(a) System Architecture (b) LSCs (Universal Chart)

Fig. 2. Car2X Example Scenario.

evaluates to true when the *TBM* sends a speed suggestion to the DIC module and the suggested speed is greater than 50. At this speed, the car is considered to be out of town, where additional services are available. Now the main chart has to match. First the speed suggestion is saved in the variable *DIC.sS*. Then the *DIC* has to send an activate message to the *EM* and is allowed to request a service, whereby, the variable *sS* has to be sent as payload. In contrast, signatures that describe obligations can be modeled as an *existential chart*, an LSC without a prechart.

To show how LSCs satisfy the requirements evaluated in the previous section, we will use existential charts for simplicity. In all examples in Fig. 3 single messages can be also considered as complex patterns.

Patterns that describe the type and order of events, are depicted in Fig. 3a) to e). The LSC in a) shows a *sequence* of events that have a fixed partial order on every lifeline and between the sending and receiving of a message. The *TBM* first sends an authentication status to the display of the *DIC*, followed by the operator name of the toll bridge and the result of the negotiation. The next chart describes a *conjunction*, where two messages can be sent in an arbitrary order. To realize this, we have to introduce a construct available for message sequence charts, but missing for LSCs – the *par* fragment. This describes a concurrent occurrence of patterns and will be reused to satisfy further requirements. The property of *negation* is modeled in c) as a *forbidden* fragment. This can be associated with the whole chart as used in the example or limited to a sub chart. So a message to display the time is not allowed during the modeled signature. In Fig. d) a *disjunction*, where only one of the messages is allowed to be sent, is depicted. Therefore, two sub charts are combined to one or-structure. Finally, in subfigure e) the *par* fragment is used to model a *simultaneous* occurrence of two messages. After the messages are sent by *DIC* the time is stored and, afterwards, evaluated by a *hot* condition. Because CAN buses do not allow sending two or more messages at the same time, a time interval that has to be less than 100 ms is tested.

Beside these patterns, a behavioral signature language has to be able to express different kinds of iterations as depicted in Fig. 3f) to h). To model iterations with the properties *exact*, *at least*, and *at most*, standard *loop* fragments of LSCs

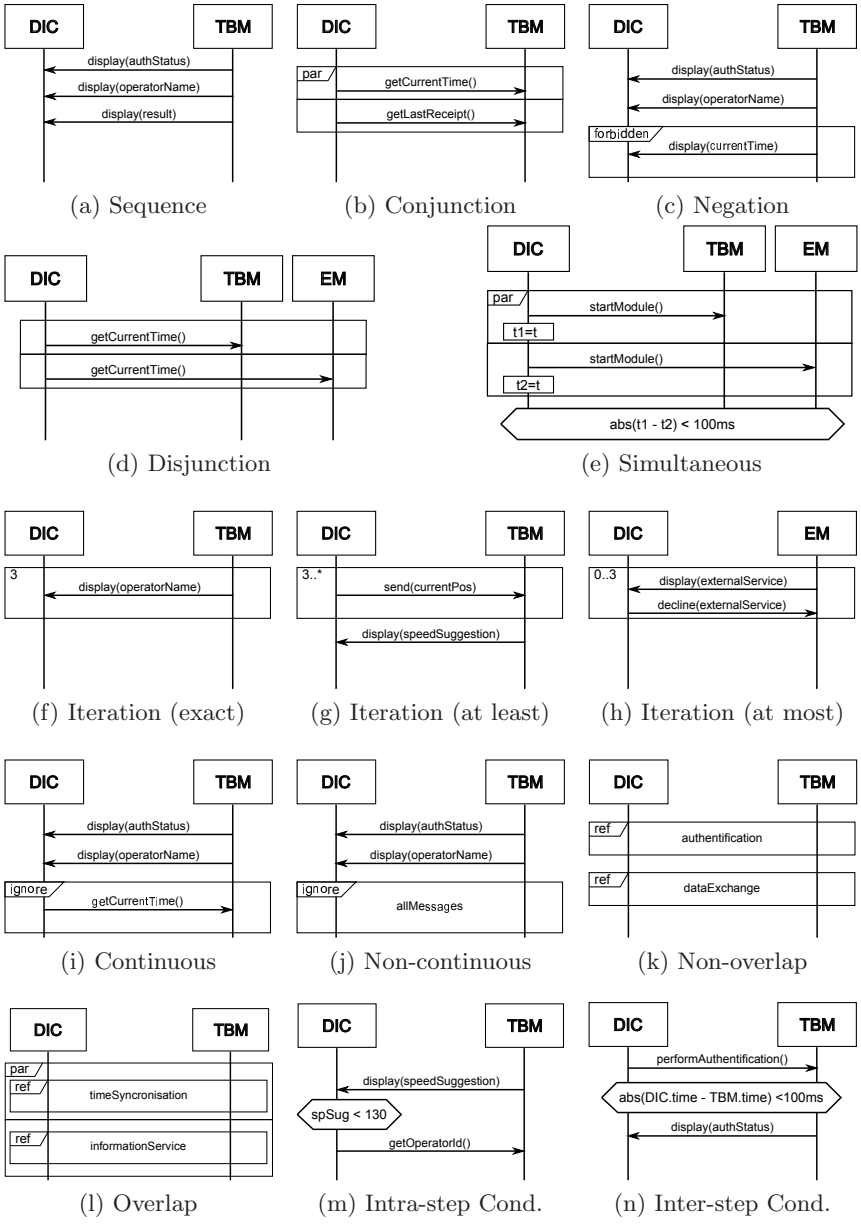


Fig. 3. Required Patterns as Live Sequence charts.

are used. Thereby, the annotations n, $n..^*$, and $n..m$ define the lower and upper bound of repetitions.

When using a signature language, it is important to define how the modeled signatures have to be interpreted. One aspect is the kind of pattern matching that can be *continuous* or *non-continuous*. In the continuous case, all messages occurring in the system during the monitoring of the signature have to be explicitly modeled. Therefore, we have introduced an additional *ignore* fragment that can be used similar to the *forbidden* fragment, but describes messages that are not relevant to the signature and can be ignored. In this way, signatures can be modeled concisely, without allowing every message that is not explicitly modeled as in the non-continuous case. To express that all additional messages should be ignored, an *ignore* fragment can be used, marked with "allMessages". Another aspect is, if two or more patterns can occur concurrently or one after the other. Figure 3k) and l) show how this is realized using the LSC syntax. For *non-overlapping* sequences the patterns are placed in subcharts – here references to other LSCs or for *overlapping* sequences a *par* fragment is used.

The ability to distinguish between *intra-step* and *inter-step* conditions is another essential concept for signature languages. An example for an *intra-step* condition is presented in subfigure m). There, the received suggested speed is checked to be less than 130 before the next message is allowed to occur. A more global condition is the *inter-step* condition, depicted in n), which compares the time on both instances with each other.

The *matching rules* – *first*, *last*, and *all* – can be modeled as single messages or by using loops. To describe a signature that matches *all* occurrences of an event type, can be specified as loop fragment – shown in Fig.3 f) to h). A single message as depicted in a) is used to match the *first* and a loop followed by a different event type is used to match the *last* occurrence.

For the *type of consumption* we decided to model signatures that are matched in a *consuming* manner, because this fits to the standard LSC syntax. The matching of a signature consumes the occurred events.

As presented in this section, we were able to satisfy all requirements that have been postulated by us and the referred authors in Sect. 3. To accomplish a compact modeling of signatures with LSCs, we had to extend the language by two new concepts, that fit the spirit of LSCs. These are the *par* fragment to describe concurrent patterns and the *ignore* fragment to achieve a compact description of signatures in a continuous pattern matching scenario.

5 Conclusion and Future Work

In this paper, we have shown that Live Sequence Charts already cover most of the crucial properties that a behavioral policy language must possess. Additionally, we have proposed some extensions that are needed to satisfy all the postulated requirements stated in Sect. 3. These extended LSCs are used in our proposed development process for security monitors as an abstract behavioral signature

language in the requirements phase. With these extensions, it is possible to model policies in an easy and even to non-practitioners understandable way.

In the future, the LSC descriptions have to be extended by timing constraints, as they are described, e.g., for UML Sequence charts in the MARTE profile. We are currently working on a prototype case tool that supports the whole process from the modeling of policies in the requirements phase until the automatic generation of monitors for different target platforms. It is based on the UML modeling tool Sparx Systems Enterprise Architect that is tailored for the modeling of all diagram types included in our process. Thereby, model-to-model transformations are specified by graph transformations by the meta-modeling tool MOFLON[5].

ACKNOWLEDGEMENTS

This work was supported by CASED (http://www.cased.de).

References

1. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(1), 45–80 (2001)
2. Giarratano, J., Riley, G.: Expert Systems: Principles and Programming, Third Edition. Course Technology (1998)
3. Groll, A., Ruland, C.: Secure and Authentic Communication on Existing In-Vehicle Networks. In: Proc. of IEEE IV'09. pp. 1093–1097 (2009)
4. Harel, D., Maoz, S., Segall, I.: Some Results on the Expressive Power and Complexity of LSCs. Lecture Notes in Computer Science 4800, 351 (2008)
5. Hussein, M., Zulkernine, M.: UMLintr: A UML Profile for Specifying Intrusions. In: Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on. pp. 8–288. IEEE (2006)
6. Jürjens, J.: UMLsec: Extending UML for Secure Systems Development. In: UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language. pp. 412–425. Springer-Verlag, London, UK (2002)
7. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., et al.: Experimental Security Analysis of a Modern Automobile. In: Security and Privacy (SP), 2010 IEEE Symposium on. pp. 447–462. IEEE (2010)
8. Kumar, S.: Classification and Detection of Computer Intrusions. Ph.D. thesis, Purdue University (1995)
9. Lindqvist, U., Porras, P.: Detecting Computer and Network Misuse through the Production-based Expert System Toolset (P-BEST). In: Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on. pp. 146–161. IEEE (2002)
10. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: UML 2002 – The Unified Modeling Language. vol. 2460/2002, pp. 426–441. Springer Berlin / Heidelberg (2002)
11. Massacci, F., Naliuka, K.: Towards Practical Security Monitors of UML Policies for Mobile Applications. In: Proc. of IEEE POLICY '07. pp. 278–278 (2007)

[5] MOFLON meta-CASE tool: www.moflon.org

12. Papadimitratos, P., Buttyan, L., et al.: Secure Vehicular Communication Systems: Design and Architecture. IEEE Commun. Mag. 46(11), 100–109 (2008)
13. Patzina, L., Patzina, S., Piper, T., Schürr, A.: Monitor Petri Nets for Security Monitoring. In: Proc. of S&D4RCES (2010)
14. Schmerl, S.: Entwurf und Entwicklung einer effizienten Analyseeinheit für Intrusion-Detection-Systeme. Diplomarbeit, Lehrstuhl Rechnernetze, BTU Cottbus (2004)
15. Sindre, G., Opdahl, A.L.: Capturing Security Requirments through Misuse Cases. In: NIK 2001 (2001), http://www.nik.no/2001
16. Sloman, M., Lupu, E.: Security and Management Policy Specification. Network, IEEE 16(2), 10–19 (2002)
17. Smith, S., Beaulieu, A., Phillips, W.G.: Modeling Security Protocols Using UML 2. Workshop – Modeling Security'08 (2008)
18. Solhaug, B., Elgesem, D., et al.: Specifying Policies Using UML Sequence Diagrams–An Evaluation Based on a Case Study. In: Proc. of IEEE POLICY '07. pp. 19–28 (2007)
19. Vigna, G., Eckmann, S., Kemmerer, R.: The STAT Tool Suite. In: DARPA Information Survivability Conference and Exposition, 2000. Proc. of DISCEX'00. vol. 2, pp. 46–55. IEEE (2002)
20. Westphal, B., Toben, T.: The Good, the Bad and the Ugly: Well-Formedness of Live Sequence Charts. In: Fundamental Approaches to Software Engineering. vol. 3922/2006, pp. 230–246. Springer Berlin/Heidelberg (2006)