Monitor Petri Nets for Security Monitoring[*]

Lars Patzina
CASED
Darmstadt, Germany
lars.patzina@cased.de

Sven Patzina
Real-Time Systems Lab
TU Darmstadt, Germany
sven.patzina@cased.de

Thorsten Piper
CASED
Darmstadt, Germany
thorsten.piper@cased.de

Andy Schürr
Real-Time Systems Lab
TU Darmstadt, Germany
andy.schuerr@cased.de

ABSTRACT

In our integrated model-based development process for security monitors, we use Live Sequence Charts (LSCs) as expressive, formal specification. Generating target specific monitors from these, requires a complex interpretation of their syntax and semantics. In this paper, we propose a Petri Net dialect as an intermediate language for monitor generation—named Monitor Petri Nets (MPNs). It is based on standard Petri Nets that are syntactically and semantically extended to suit the needs of monitoring. With our MPNs, we are able to represent use and misuse cases described by LSCs in a format that is easy to interpret. MPNs provide the basis for the generation of SW/HW security monitors or can alternatively be interpreted by a generic monitor.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Security, Languages

Keywords

Live sequence charts, monitor petri nets, signature modeling

1. INTRODUCTION

Driven by technical innovation, embedded systems become increasingly interconnected. Prominent examples can be found in the automotive domain, where, after the deployment of toll collection and active road sign technologies, Car-to-Car (C2C) and Car-to-Infrastructure (C2I) communication—generally denoted as Car-to-X (C2X)—are emerging.

[*]This work was supported by CASED (www.cased.de).

Since embedded systems were originally designed as closed systems in many areas, often little attention has been paid to implementing security measures, such as encryption and safe component design, impeding attacks from the outside world.

In order to retroactively secure such systems against external adversaries, new areas of research were established, as the authors of [12] claim. These address issues such as the design of secure vehicular communication and the development of architectures providing enhanced privacy and security. The necessity for additional security measures is further emphasized by [7], who demonstrates that modern intra-vehicular networks are highly vulnerable to passive and active attacks. The main identified security problems are the lack of authentication between components and the coupling of different busses via gateways, enabling attackers to passively evesdrop on communication with manipulated electronic control units (ECUs).

Just as detecting all security flaws of a component during design time is rarely possible, so is the consideration of all possible attack scenarios. Furthermore, redesigning or refactoring existing components towards enhanced security awareness is often economically or technically infeasible, especially in large and heterogeneous systems. Therefore, our system model inherently presumes the insecurity of system components, either due to unknown vulnerabilities, or due to the required integration of legacy components.

[10] has shown that system monitoring aids to detect intrusions, which utilize previously unknown attacks and faults, by profiling system behavior at run-time. The two most common approaches for monitoring are, on the one hand, signature detection, featuring a low false-positive rate and high effectiveness against attacks that are similar to known attacks and attack classes, and, on the other hand, anomaly detection. In contrast to signature-based detection schemes, anomaly detection aims at detecting behavioral anomalies. By employing statistical profiling techniques, anomaly detection schemes are potentially able to reveal unknown attacks by detecting their impact on the system behavior. Unfortunately, this ability has the price of high false-positive rates, therefore requiring either user interaction to evaluate the threat or computationally cheap self-healing techniques to automatically reset the system to a known safe and secure state.

Our goal is a comprehensive, model-based security engineering process based on the Model Driven Architecture

(MDA) concept to automatically generate monitors, depicted in Figure 1. In this process, the intended system behavior, as well as known attack patterns and attack classes, are described and incorporated early in the development process, during the requirements engineering phase. For specification, we employ use and misuse cases [14], described in more detail with Live Sequence Charts (LSCs), a more expressive extension of Message Sequence Charts (MSCs). By an automatic transformation and enrichment with system-specific information, these specifications are translated to security monitors. The combination of (mis-)use cases and LSCs enables the developer to model the intended system behavior as well as potential attack patterns in a compact and comprehensible way. However, a direct generation of monitors from LSCs requires a complex interpretation of the LSC semantics.

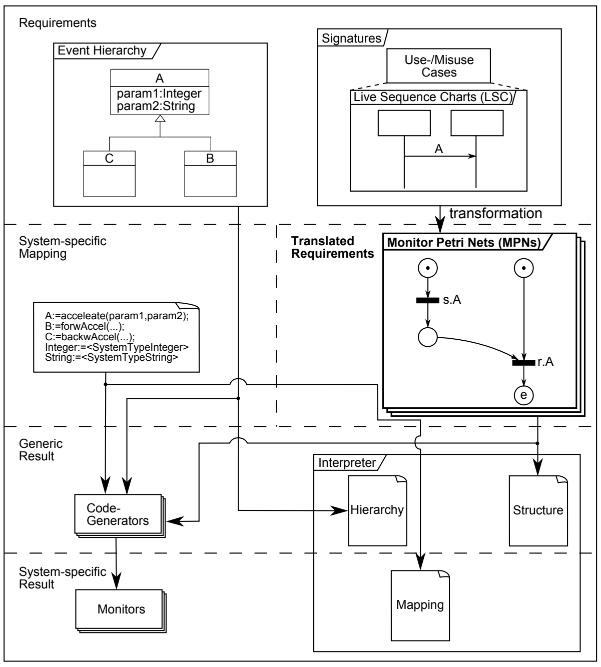

Figure 1: Model-based security engineering process.

In this paper, we propose Monitor Petri Nets (MPNs), a novel extension to regular Petri nets, tailored towards a compact representation of use case and misuse case models. By employing MPNs, we are able to trace the behavior of large heterogeneous systems, consisting of many concurrently and asynchronously communicating subsystems. LSCs can now be efficiently compiled into security monitors, by first translating them into MPNs—incorporating ideas from [3]—and afterwards implementing the MPNs in software or reconfigurable hardware, using e.g. SystemC as the target language.

This paper is structured as follows. In Section 2, we introduce a running example scenario where Live Sequence Charts are used to describe a simple DoS attack scenario as (mis-)use case. Section 3 covers related work and depicts why a new class of Petri nets is needed to implement our approach. In Section 4, we define MPNs and their execution semantics. Afterwards, we present the application of the MPN language to the example in Section 5. Section 6 concludes the paper and gives an outlook on future work.

2. EXAMPLE SCENARIO

In this section, we present the running example used in the paper to illustrate the Monitor Petri Net language, introduced in Section 1. We assume a simplistic toll bridge scenario as shown in Figure 2, where cars approach and connect to a road side unit (RSU) as they enter its communication range (depicted by a dotted circle).

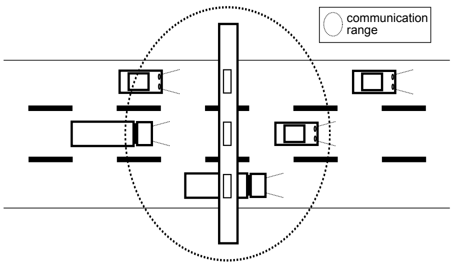

Figure 2: Example toll bridge scenario.

To establish a connection, the communication protocol requires a simple *connect* and *ack* handshake. A car transmits up to three payload packets, *data_x* and the optional *data_y* and *data_z*. After that, it terminates the connection by sending a *disconnect* packet. To track the number of active connections, a global counter variable *ac*, which is shared among all instances of corresponding sequence charts, is incremented every time a connection is established and decremented after its termination.

Figure 3a depicts a use case description of this protocol, modeled as Live Sequence Chart (LSC). LSCs [4] are an extension of the common Message Sequence Charts (MSCs), offering a distinction between *hot* and *cold* elements. The hot message *data_x*, depicted as a solid arrow, has to occur, whereas *cold* messages like *data_y*, depicted by a dashed arrow, can occur. The difference to an asynchronous message located in an optional MSC fragment is that a message needs not to be received before the next message from the same sender can be sent. By matching the use case pattern against communications occurring at run-time, deviations from the use case and consequently a misconduct of the system can be detected.

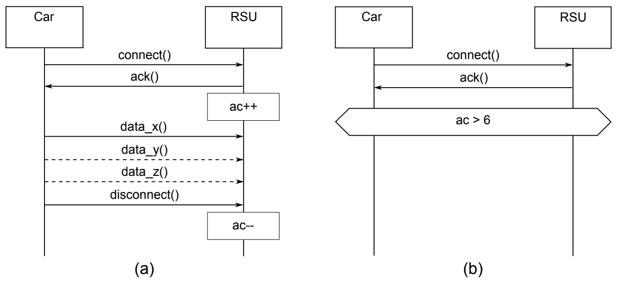

Figure 3: Example LSCs: (a) use, (b) misuse case.

Misuse case descriptions complement the use case descriptions, as they can be used to model unintended behavior or presumed attacks. Figure 3b shows such a misuse case, which describes a simple pattern that is capable of detecting a kind of a Denial of Service (DoS) attack against the

RSU. In this example, we assume a physical limitation of the communication range of the toll bridge, implying that only a certain amount of vehicles can concurrently be connected at a given time. Therefore, we permit a hard limit of simultaneous connections ($2{*}\#lanes = 6$) and presume a DoS attempt when this limit is exceeded. In order to check this condition, the global variable ac is compared against the number of allowed connections, each time the toll bridge acknowledges a connection request. When the condition $ac > 6$ is met, the modeled misuse case is detected, otherwise the misuse case has not occurred. The specification of countermeasures is out-of-scope of this paper.

The introduced examples will be used to demonstrate the application of the Monitor Petri Net language that will be motivated in the next section.

No.	*Requirements*	[16]	[3]	[5]	[9]
		$SD \rightarrow FSM$	$LSC \rightarrow CPN$	$SD \rightarrow CPN$	$PLSC \rightarrow LTL$
	Exec. monitor/repres.	-/+	-/+	-/+	-/+
1	Separate send/receive	-	-	-	-
2	Interleaving sequences	o	+	+	+
3	Cold/optional elements	o	+	o	+
4	Compact	-	o	o	-
5,6	Global/local variables	-	+	+	+
7	Timing constraints	-	+	+	+
8	Positive/negative ends	-	+	-	-
9	Deterministic firing	-	-	-	-
10	Eff. simult. monitoring	-	-	-	-

+ (good); o (average); - (bad/not addressed)

Table 1: Transformations to state-based languages.

3. RELATED WORK

Our approach to an automatic and model-driven generation of monitors is based on the concept of use and misuse case modeling. Misuse cases [14] were introduced as a description for security requirements, being more comprehensible than those used in existing standards for security engineering – like the Common Criteria or the ECMA standards. Alexander describes misuse cases as "a use case from the point of view of an actor hostile to the system under design" [1]. In [2], he further states that misuse cases are an effective tool for modeling potential attacks against a system.

In order to model use and misuse cases, a variety of model-based languages can be used, e.g. UML Sequence Diagrams (SDs), Message Sequence Charts (MSCs), or Live Sequence Charts (LSCs). All of them describe sequences of messages over time, but differ in expressiveness and complexity. MSCs and SDs are similar to each other, whereas LSCs extend MSCs by the distinction between elements that may or that must occur.

Solhaug et al. [15], Massacci and Naliuka [11] use UML 2.0 SDs to describe model-based policies. They show that the usage of SDs is suitable for this purpose, but also identify the lack of formal semantics as a major drawback, because it restricts the models' expressiveness. To resolve this shortcoming, Massacci and Naliuka suggest to enrich SDs with Linear Temporal Logic (LTL) formulas, whereas Solhaug et al. propose to use SDs in ways, which "do not conform with the spirit of UML". As an alternative to his approach, Solhaug recommends taking a closer look at LSCs, which can be utilized to capture deontic constraints in the model. Following this idea, Kumar et al. [9] use LSCs for the specification and verification of protocols, because LSCs are more expressive and semantically richer than MSCs or SDs.

For monitoring, analysis and verification, MSCs or LSCs are usually translated to other formalisms, such as Finite State Machines (FSMs) and Petri nets (PNs). In [16], Whittle et al. were the first to describe an approach to model security concerns as executable misuse cases, avoiding often used informal notations. They extend Extended Interaction Overview Diagrams (EIODs) by the concept of misuse cases. To execute their models, Whittle et al. use a tool named UCSIM [8] that generates an FSM for each participant of every single sequence diagram. In order to synchronize all FSMs, the translation requires the insertion of synchronization events, which is the major disadvantage of this process. These events produce a considerable overhead and reduce readability, because the logic of the SDs is not explicitly stated in the FSMs.

Transformations from models to Petri nets have mostly been developed to be able to analyze and verify the correctness of the source model. Amorim et al. [3] have proposed such a transformation. They transform LSCs to the more versatile and expressive Colored Petri Nets (CPNs) to analyze the system's behavior by simulating it.

Fernandes et al. [5] consider a transformation from UML 2 use cases, described by SDs, to CPNs, with the goal to obtain an executable model that is easier to understand than SDs. To translate all concepts of the SDs, he makes extensive use of the ML language, a functional programming language, and uses hierarchical structures to get a "readable" result.

Kumar et al. transform protocols described as LSCs in temporal logic [9]. As the transformation to temporal logic leads to an explosion of the temporal logic formula, it is not usable for our approach.

Not all the transformations from SCs to FSMs and PNs, introduced in this section, are equally suited to model security-aware system monitors, as their feature sets are quite different. Table 1 lists the presented approaches and evaluates how well each of them satisfies our demands on an intermediate representation.

The comparison reveals that FSMs are an inappropriate representation for monitoring systems described by LSCs, as the expression of some of the key concepts, such as asynchronous messages (1), interleaving concurrently executed subcommunication threads (2) and optional/cold message exchange sequences (3), is either difficult or complex in FSMs and, therefore, not compact (4). Modeling all possibilities that arise from optional constructs in SDs or even cold constructs in LSCs, leads to an exponentially growing number of states and is therefore infeasible. Even statecharts with parallel sub-states are not well suited to capture the semantics of LSCs, because of their limited parallel modeling concept. Consequently, all variants of FSM-based approaches like the transformation of SDs to FSMs [16] are inappropriate for our purpose.

Therefore, we decided to model the specifications of security-relevant system protocols as use and misuse cases with LSCs and translate them into an appropriate variant of Petri nets. These Petri nets must support manipulation of local and global variables (5), definition of transition predicates over their variables (6), and representation of timing properties (7). Furthermore, we have to distinguish between permitted and forbidden end states (8) that signal recognition of a communication pattern related to a use or misuse case. Last but not least, an execution semantic is needed where transitions fire whenever possible (9) and where many concurrently running communication instances can be manipulated efficiently (10). To the best of our knowledge, all existing PN-based approaches that are used for the specification of security monitors violate at least one or more of the requirements mentioned above.

One of these approaches is published by Frankowiak et al., who use Petri nets to specify a low-cost process monitor on a microcontroller [6]. Therefore, they enhance regular Petri nets by a token generator and end places (bins), and introduce subnets linked by a control net. Their proposed Petri net dialect e.g. meets requirements 6, 7, and 10, but misses 5, 8 and 9.

Another approach is the Event Description Language (EDL)[13], which is presented as a holistic signature language for intrusion detection systems. It is based on place/transition nets with its own semantics. Although this language suites better for our concerns, a concept for global variables (5) and timers (7) is missing. Furthermore, the standard execution semantics of EDL is non-continuous (4), i.e., there can be additional events between the specified ones, and, therefore, does not fit well to the semantics of LSCs. Hence, the modeling of continuous signatures results in many escape places. Moreover, the authors do not provide a formal definition of EDL.

As none of the existing approaches complies with all the requirements, we adopted place/transition nets as a conceptual foundation and extended their definition to satisfy all requirements. We present a simplified version of the definition of this novel Petri net dialect—named Monitor Petri Nets—in the following section.

4. MONITOR PETRI NET DEFINITION

After defining the requirements of an appropriate language in Section 3, we introduce a syntactically and semantically modified definition of Petri nets, named Monitor Petri Nets (MPN). Due to space restrictions, we will exclude the formal definition of the tokens that can carry values like in CPNs. Furthermore, we omit a language for actions and functions at the transitions.

4.1 Syntax definition

In comparison to standard Petri nets, the MPN language defines special start and terminal places. Every start place generates tokens with a generation tag. This enables us to monitor more than one matching attempt of a communication pattern in a single instance of an MPN. When a token reaches a terminal place, the execution for all tokens with the same generation is stopped and the monitoring result is evaluated. The syntax of the MPN is presented in Definition 1.

Definition 1. *A net $mpn \in MPN$ is an extended variant of a place/transition net with $mpn = (S, S_i, S_t, T, F, p, m)$ where*

S is a finite set of places $S = \{s_1, s_2, \ldots, s_{|S|}\} \neq \emptyset$.

$S_i \subset S$ is the set of all initial places that contains tokens in the initial state of the net.

$S_t \subset S$ is the set of all terminal places that end the execution of the net when a token reaches one of them.

T is the finite set of transitions $T = \{t_1, t_2, \ldots, t_{|T|}\} \neq \emptyset$.

$S_i \neq \emptyset$; $S_t \neq \emptyset$; $S_i \cap S_t = \emptyset$; $S \cap T = \emptyset$.

$F \subseteq (S \setminus S_t \times T) \cup (T \times S \setminus S_i)$ is the flow relation that connects places and transitions.

$p : T \times C \to \{true; false\}$ is a predicate that assigns a logical value to every transition for a given external context.

$m : S \times \mathbb{N} \to \{1, 0\}$ is the (initial) marking of the net that defines network configurations. The function distinguishes different generations of tokens that can be identified through natural numbers. A place s can hold at most one token of each generation g.

preset: $\bullet t = \{s \in S | (s,t) \in F\}$.

postset: $t\bullet = \{s \in S | (t,s) \in F\}$.

4.2 Execution Semantics

For our monitoring purpose, the execution semantics of deterministic Petri nets is not appropriate. It is necessary to model descriptions as compact as possible and allow a simple transformation between LSCs and MPNs. Therefore, we define an execution semantics based on micro and macro steps, similar to the formal definition of statecharts presented by Harel. The resulting MPN execution semantics fires all enabled transitions simultaneously. Different enabled transitions with overlapping presets even share consumed tokens when firing concurrently in one macro step.

Definition 2 (Initial Marking). *In all initial places, a token with generation $g = 1$ is created.*

$$m_0(s,g) = \begin{cases} 1 & \text{for } s \in S_i \wedge g = 1 \\ 0 & \text{else.} \end{cases} \tag{1}$$

Definition 3 (Activation Condition). *A transition is enabled if all places in its preset carry a token of generation g. Additionally, its predicate has to evaluate to true.*

$$enabled(t,c,g) :\Leftrightarrow p(t,c) \wedge \forall s \in \bullet t : m(s,g) = 1. \tag{2}$$

When all enabled transitions T for a context c of an MPN are considered, the transition from a Petri net mpn to a new Petri net mpn' is named macro step.

In detail, a new marking m' of the MPN is derived from m in every macro step. Therefore, the current configuration is frozen and used to derive the new configuration after the macro step. Every step is evaluated on the first configuration and the result is stored in the new one. With that rule, there is no competition between two or more transitions that are enabled in a macro step. Each place can only store one token of a generation, while another token of the same generation is merged with an already existing token.

Definition 4 (Macro Step Execution). *For each generation of tokens all enabled transitions fire.*
$macro : MPN \times C$ is defined as follows:

```
$macro(m,c) =$
  $max\_g' = max\_g$                //maximal generation
  foreach $s \in S, g \in \mathbb{N}$          //copy net
    $m'(s,g) = m(s,g)$;
  foreach $t \in T$, $g \in \mathbb{N}$ with $enabled(t,c,g)$
    foreach $s \in \bullet t$              //process preset
      $m'(s,g) = 0$;
      if ($s \in S_i \wedge g = max\_g$) then
        $max\_g' = max\_g + 1$; //inc. max. gen. number
```

```
foreach $t \in T$, $g \in \mathbb{N}$ with $enabled(t, c, g)$
  foreach $s \in t\bullet$          //process postset
    $m'(s, g) = 1$;
   if ($max\_g' > max\_g$) then
     foreach $s \in S_i$          //create new gen. tokens
       $m(s, max\_g') = 1$;
$fired[] = 0$                     //init array (0)
foreach $t \in T$, $g \in \mathbb{N}$ with $enabled(t, c, g)$
  $fired[g] = 1$;                 //log performed steps
$postprocessing(m', fired)$;
return $m'$;
```

When all transitions have fired, a post processing step is done to treat tokens that have reached a terminal place and generations whose configuration has not changed.

Definition 5 (Postprocessing). *For all generations with at least one token in a terminal place or the generations for which no transition has fired, the configuration of m' is evaluated. Afterwards, all tokens of this generation are deleted in m'.*

```
$postprocessing(m', fired) =$
  foreach $s_t \in S_t$, $g \in \mathbb{N}$
    if $m'(s_t, g) = 1$
      $evaluateMonitoringResult(g)$;// check term. places
        foreach $s \in S$
          $m'(s, g) = 0$;
  foreach $g \in \mathbb{N}$
    if $fired[g] = 0 \wedge relevantEvent(g)$
      $evaluateMonitoringResult(g)$;// check firing
        foreach $s \in S$
          $m'(s, g) = 0$;
  return $m'$;
```

The *evaluateMonitoringResult* method checks the current configuration of the MPN and determines, which action should be performed to reestablish a stable state of the affected components. These actions can be defined as a mitigation of the corresponding misuses case. When a generation of tokens has not been used by a firing transition for a relevant event, e.g. sent by the instance associated with the currently processed generation, the described pattern has not been matched. For a use case, this implies that a failure has occurred and for a misuse case that it has not been detected. To reduce the amount of false-positives due to the continuous matching semantics, i.e., every event is taken into account, of the MPNs, we extend the LSCs by so called *ignored messages* that are represented in the MPN as self-transitions or a parallel MPN. The *relavantEvent* method evaluates to true, when an event is sent from the instances of the corresponding generation or when a transition without an event is enabled.

5. APPLICATION OF MPN LANGUAGE

In this section, we show how the defined MPN language can be used to describe the use and the misuse case introduced in Section 2. We focus on the crucial concepts of the transformation process from LSCs to the MPNs. In Figure 4, the corresponding MPNs for our example are shown.

The graphical MPN representation consists of four types of places that are explained in Table 2, transitions that are annotated by [*guard*]*event*/*action*, and arcs alternately connecting places and transitions. In the transformation process, a start place is created for each lifeline in the LSC.

Type	*Symbol*	*Description*
StartPlace	(·)	Each StartPlace holds a token when a new instance of the net is created.
Place	( )	A standard Petri net place that can hold one token of each generation.
EndPlace	(e)	When an EndPlace is reached, a permitted pattern is detected.
FailurePlace	(f)	When this place is reached, a modeled attack (misuse) or a mismatch of the expected behaviour is detected.

Table 2: Types of places.

		Places of (mis-)use case								
#	*Inst.: Ev.*	*1a*	*1b*	*2*	*3a*	*3b*	*4*	...	*f*	*ac*
0	...									5
1	Car6: s.con	6	6							5
2	Car7: s.con	6,7	6,7							5
3	Car6: r.con	7	6,7	6						5
4	Car7: r.con		6,7	6,7						5
5	Car6: s.ack		6,7	7	6	6				6
6	Car6: r.ack		7	7		6	6			6
7	Car7: s.ack		7		7	7				7
8	Car7: r.ack					7	7			7
9	...								7	7

Table 3: Example monitor execution.

Each location, i.e., a point on a lifeline touched by an element, is represented by a separate place in the MPN. Every asynchronous message of the LSC is translated to one transition for the sender and one for the receiver side. These transitions are interconnected by a place that succeeds the sender transition. Hence, a message has to be sent before it can be received. Cold messages result in by-pass transitions.

We demonstrate the execution semantics by stimulating the MPNs with a sequence of events, presented in Table 3. The first column represents the steps of execution, the second the events that occur, the third shows which place holds a token and the last represents the number of active connections. For a compact representation, the places $1a$ to 4 apply to both, use and misuse MPN, f only to the misuse case. Each token is depicted by its generation number. We assume that in step 0, five cars are already connected ($ac = 5$) to the toll bridge (RSU) but their tokens are neglected. In the first four steps *Car6* and *Car7* try to connect to the RSU. Afterwards, in step 5 and 6 the connection attempt of *Car6* is acknowledged and ac is incremented to 6. With the next message, the transition with the guard $[ac > 6]$ of the misuse case does not fire for the tokens with generation 6—no violation detected and the tokens are discarded. When the RSU establishes the connection to *Car7*, ac increases to 7 and the following event or timeout generates a token in the failure place f—the misuse case is detected.

Figure 4a shows the LSC and the resulting MPN of the more complex use case presented in Section 2 side by side. Every message, even the cold (optional) messages, can be directly mapped from the LSC to the MPN representation. The state of LSC during execution (active cut) is directly represented by the state of the MPN. A violation of a use case is detected when an event associated with involved participants does not fire a transition in the MPN.

For monitoring a single instance, the generated MPNs can be split into two regions. The left shows the behavior of

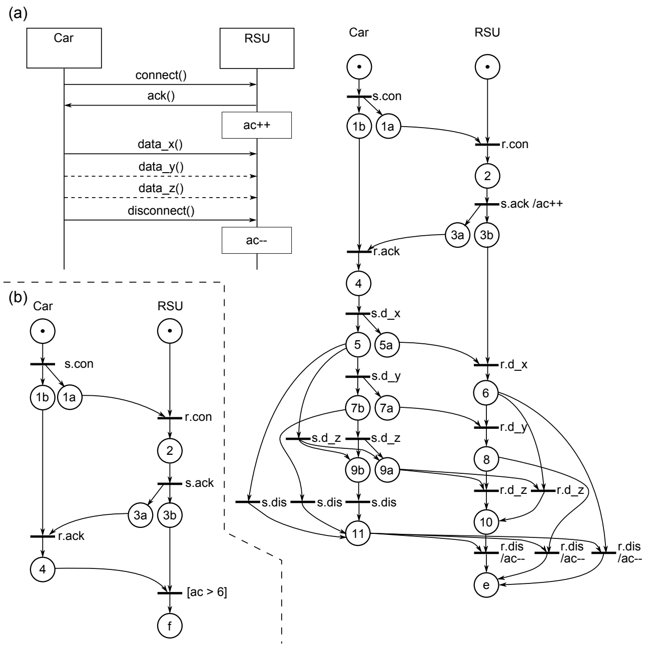

Figure 4: Example MPN: (a) use, (b) misuse case.

instance *Car* and the right shows instance *RSU* that are coupled by synchronization places, depicted by an *a* in their name. By removing these places and their adjacent arcs, we receive MPNs for the individual lifelines.

6. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a formal definition of a novel Petri net dialect, called Monitor Petri Nets (MPNs), that constitutes a simple, compact, and easy to process representation of use and misuse cases described by Live Sequence Charts (LSCs). In comparison to the immediate operation on LSCs, the interpretation of MPNs by a security monitor, or the generation of executable monitoring code from MPNs, is considerably simplified. The transformation from LSCs to MPNs can be performed automatically by applying a small set of rules.

MPNs are used as an intermediate format in a holistic development process for security monitors, as introduced in Section 1, which is not restricted to the security domain or C2X scenarios in particular. Due to the expressiveness of LSCs and the preserved semantics in MPNs, our approach can be adopted to other monitoring tasks like safety or process monitoring. Furthermore, it can be utilized as the basis for generating an oracle used for testing and verification.

In future work, we will use, in contrast to our DoS attack example, outlined in Section 2, the full concept of LSCs to model the allowed and prohibited behavior of the system in a more precise and formal way. As the transformation from LSCs to MPNs has to be formally defined and implemented, we target a model-based approach employing the meta-modeling tool MOFLON[1]. Based on the intermediate MPN format, an automatic generation of software or hardware components, comprising system specific information, will be performed. Additionally, a monitor realized as an interpreter for the XML exchange format as proposed by the ISO/IEC 15909-2 standard is conceivable. To ensure the correctness of the behavioral description, it is necessary to investigate how standard metrics of Petri nets like lifeness can be adopted for the verification of MPNs.

[1]MOFLON homepage: http://www.moflon.org

7. REFERENCES

[1] I. Alexander. Misuse cases: Use Cases with Hostile Intent. *IEEE Software*, 20(1):58–66, 2003.

[2] I. F. Alexander. Initial Industrial Experience of Misuse Cases in Trade-Off Analysis. *Proc. of IEEE RE'02*, pages 61–68, 2002.

[3] L. Amorim, P. Maciel, et al. Mapping Live Sequence Chart to Coloured Petri Nets for Analysis and Verification of Embedded Systems. *SIGSOFT Softw. Eng. Notes*, 31(3):1–25, 2006.

[4] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[5] J. M. Fernandes, S. Tjell, et al. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *Proc. of IEEE SCESM '07*, page 2, 2007.

[6] M. R. Frankowiak, R. I. Grosvenor, et al. Microcontroller-Based Process Monitoring Using Petri-Nets. *EURASIP J. Embed. Syst.*, pages 1–12, 2009.

[7] A. Groll and C. Ruland. Secure and Authentic Communication on Existing In-Vehicle Networks. In *Proc. of IEEE IV'09*, pages 1093–1097, 2009.

[8] P. Jayaraman and J. Whittle. UCSIM: A Tool for Simulating Use Case Scenarios. In *ICSE COMPANION '07*, pages 43–44, 2007.

[9] R. Kumar, E. Mercer, et al. Improving Translation of Live Sequence Charts to Temporal Logic. *ENTCS'09*, 250(1):137–152, 2009.

[10] S. Kumar. *Classification and Detection of Computer Intrusions.* PhD thesis, Purdue University, 1995.

[11] F. Massacci and K. Naliuka. Towards Practical Security Monitors of UML Policies for Mobile Applications. In *Proc of IEEE POLICY '07*, pages 278–278, 2007.

[12] P. Papadimitratos, L. Buttyan, et al. Secure Vehicular Communication Systems: Design and Architecture. *IEEE Commun. Mag.*, 46(11):100–109, 2008.

[13] S. Schmerl, U. Flegel, et al. Vereinfachung der Signaturentwicklung durch Wiederverwendung. In *Proc. of SICHERHEIT 2006*, pages 201–212, 2006.

[14] G. Sindre and A. L. Opdahl. Capturing Security Requirments through Misuse Cases. In *NIK 2001*, 2001.

[15] B. Solhaug, D. Elgesem, et al. Specifying Policies Using UML Sequence Diagrams–An Evaluation Based on a Case Study. In *Proc. of IEEE POLICY '07*, pages 19–28, 2007.

[16] J. Whittle, D. Wijesekera, et al. Executable Misuse Cases for Modeling Security Concerns. In *Proc. of ICSE '08*, pages 121–130. ACM, 2008.