A Transformational Approach to Facilitate Monitoring of High-level Policies

Fredrik Seehusen and Ketil Stølen
SINTEF ICT / Dep. of Informatics, University of Oslo
{fredrik.seehusen,ketil.stoelen}@sintef.no

Abstract

We present a method for (1) specifying high-level security policies using UML sequence diagrams and (2) transforming high-level sequence diagram policies into low-level state machine policies that can be enforced by monitoring mechanisms. We believe that the method is both easy to use and useful since it automates much of the policy formalization process.

1 Introduction

Policies are rules governing the choices in the behavior of a system [11]. This paper focuses on a particular class of policies, namely security policies that are enforceable by the class of mechanisms that work by monitoring execution steps of some system, referred to as the *target*. This class is called EM, for Execution Monitoring [10].

Security policies are often initially expressed as short natural language statements. Formalizing these statements is time consuming since they often refer to high-level notions such as "opening a connection" or "sending an SMS" which must ultimately be expressed as sequences of security relevant actions of the target. If several policies refer to the same high-level notions, or should be applied to different target platforms, then these notions must be rewritten for each new policy and each new target platform.

Clearly, it is desirable to have a method that automates as much of the formalization process as possible. In particular, the method should: (1) support the formalization of policies at a high level of abstraction; (2) offer automatic generation of low-level policies from high-level policies; (3) facilitate automatic enforcement by monitoring of low-level policies; (4) be easy to understand and employ by the users of the method (which we assume are software developers).

The method we present has three main steps which accommodate the above requirements. *Step I:* The user of our method receives a set of policy rules written in natural language, and formalizes these using UML sequence diagrams. *Step II:* The user selects a set of transformation rules (expressed in UML sequence diagrams) from a transformation library, and applies these using a tool to automatically obtain an intermediate low-level policy. *Step III:* The tool automatically transforms the intermediate low-level policy into a UML state machine that governs the behavior of an EM mechanism.

There are two main advantages of using this method as opposed to formalizing low-level policies directly using UML state machines. First, much of the formalization process is automated due to the transformation from high to low level. This makes the formalization process less time consuming. Second, it will be easier to show that the formalized high-level policy corresponds to the natural language policy it is derived from, than to show this for the low-level policy. The reason for this is that the low-level policy is likely to contain implementation specific details which make the intention of the policy harder to understand.

The choice of UML is motivated by requirement (4). UML is widely used in the software industry. It should therefore be understandable to many software developers which are the intended users of our method. UML sequence diagrams are particularly suitable for policy specification in the sense that they specify partial behavior (as opposed to complete), i.e. the diagrams characterize example runs or snapshots of behavior in a period of time. This also means that UML sequence diagrams allow for the explicit specification of negative behavior, i.e. behavior that the target is not permitted to engage in. This is useful because the only kind of policies that can be enforced by EM mechanisms are prohibition policies.

The rest of this paper is structured as follows: Sect. 2 - Sect. 4 presents steps I - III of our method. Sect. 5 discusses related work and provides conclusions.

2 Step I: Specifying High-level Policies with Sequence Diagrams

In the first step of our method, the user (i.e. a software developer) receives a set of policy rules written in natural language. The user then formalizes these rules using UML sequence diagrams. In this section we show how to express

978-0-7695-3133-5/08 $25.00 © 2008 IEEE
DOI 10.1109/POLICY.2008.21

IEEE computer society

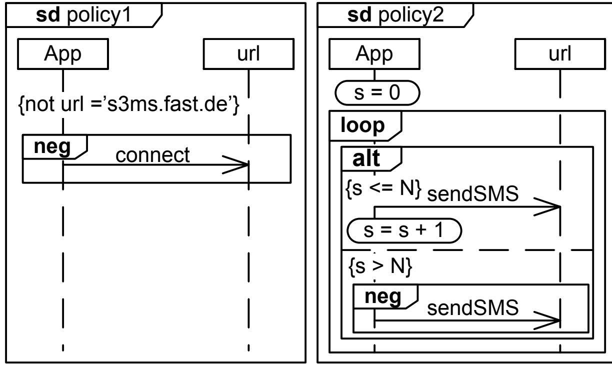

Figure 1. High-level policy 1 and 2

two security policies. The examples are taken from an industrial case study conducted in the EU project S^{3}MS (Security of Software and Services for Mobile Systems).

As the running example of this paper, we consider applications on the Mobile Information Device Profile (MIDP) Java runtime environment for mobile devices. We assume that the runtime environment is associated with an EM mechanism that monitors the executions of applications. Each time an application makes an API-call to the runtime environment, the EM mechanism receives that method call as input. If the current state of the state machines that governs the EM mechanisms has no enabled transitions on that input, then the application is terminated because it has violated the security policy of the EM mechanism.

The first policy we consider is *"The application is only allowed to establish connections to the address s3ms.fast.de."* This policy is specified by the UML sequence diagram policy1 on the left hand side of Fig. 1.

Sequence diagrams describe communication between system entities referred to as *lifelines* (represented by vertical dashed lines). An arrow between two lifelines represents a message being sent from one lifeline to the other in the direction of the arrow. The two lifelines of policy1 are App representing the target of the policy, and url representing an arbitrary address that the target can connect to. The sending of message connect from the App to url represents an attempt to open a connection.

Expressions of the form $\{b\}$ (where b is a Boolean expression) are called *constraints*. Intuitively, the interaction occurring below the constraint will only take place if the constraint evaluates to true.

The constraint in diagram policy1 should evaluate to true if url is *not* equal to the address "s3ms.fast.de" (which according to the policy is the only address that the application is allowed to establish a connection to).

Interactions that are encapsulated by the neg operator specify negative behavior, i.e. behavior which the target is not allowed to engage in. Thus diagram policy1 should be read: App is not allowed to connect to the arbitrary address url if url is different from the "s3ms.fast.de".

UML sequence diagrams are partial as opposed to complete, i.e. they explicitly describe *positive behavior* and *negative behavior*. The behavior which is not explicitly described by the diagram is called *inconclusive* meaning that it is considered irrelevant for the interaction in question.

The semantics of sequence diagrams can be expressed in terms of *traces*, i.e. sequences of events. A formal definition of the semantics of sequence diagrams can be found in e.g. [5, 8].

When using sequence diagrams to formalize prohibition policies, we are mainly interested in traces that describe negative behavior. If an application is interpreted as a set of traces, then we say that the application *adheres* to a policy if none of the application's traces have a negative trace of the policy as a sub-string. Thus we take the position that the target is allowed to engage in (inconclusive) behavior which is not explicitly described by a given policy. This is reasonable since we do not want to use policies to express the complete behavior of the target.

Turning back to the example, an application is said to adhere to the policy of Fig. 1 if none of its traces contain the transmission of the message connect to an address which is different from s3ms.fast.de.

The second natural language policy is *"The application is not allowed to send more than N SMS messages (where N is a natural number)."*

The policy is specified by the sequence diagram policy2 on the right hand side of Fig. 1. Boxes with rounded edges contain assignments of variables to values. In diagram policy2, the variable s is initialized to zero, and incremented by one each time the application sends an SMS. The loop operator is used to express the iteration of the interaction of its operand.

The alt-operator is used to express *alternative* interaction scenarios. In policy2, there are two alternatives. The first alternative is applicable if the variable s is less than or equal to N (representing an arbitrary number). In this case the application is allowed to send an SMS, and the variable s is incremented by one. The second alternative is applicable when s is greater than N. In this case, the application is not allowed to send an SMS as specified by the neg-operator.

3 Step II: Specifying Transformations with Sequence Diagrams

In the second step of our method, the user selects a set of transformation rules from a transformation library. The user then employs a tool (implemented in Prolog and Java) which automatically applies the transformation rules to the high-level policy such that a low-level policy is produced.

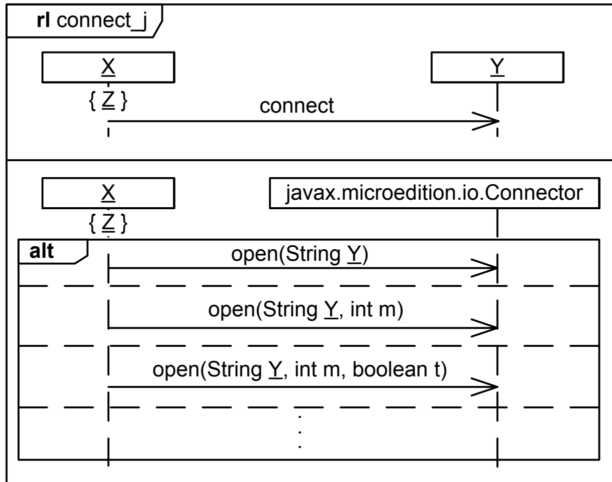

Figure 2. Transformation rule

In the following we show how a transformation to the low-level can be defined using UML sequence diagrams. An advantage of using sequence diagrams for this purpose is that the writer of the transformation rules can express the low-level policy behavior using the same language that is used for writing high-level policies.

A *transformation rule* is specified by a pair of two *diagram patterns* (diagrams that may contain *meta-variables*), one left hand side pattern, and one right hand side pattern. When a transformation rule is applied to a diagram d, all parts of d that matches the left hand side pattern of the rule are replaced by the right hand side pattern. Meta-variables are bound according to the matching. A diagram pattern dp matches a diagram d' if the meta-variables of dp can be replaced such that the resulting diagram is syntactically equivalent to d'.

The policies described in the previous section are not enforceable since the behavior of the target is not expressed in terms of API-calls that can be made to the MIDP runtime environment. Recall policy1 of Fig. 1. It has a single message connect which represents an attempt to open a connection. In order to make the policy enforceable, we need to express this behavior in terms of API-calls that can be made to the runtime environment.

Fig. 2 illustrates a transformation rule which describes how the connect message is transformed into the API-calls which are used to establish a connection via the MIDP runtime environment. The top most diagram of Fig. 2 represents the left hand side of the rule, and the bottom most diagram represents the right hand rule. In the diagram, all meta-variables are underlined.

When the rule of Fig. 2 is applied to policy1 of Fig. 1, the message connect is replaced by the right hand side pattern, i.e. the lower most diagram of Fig. 2.

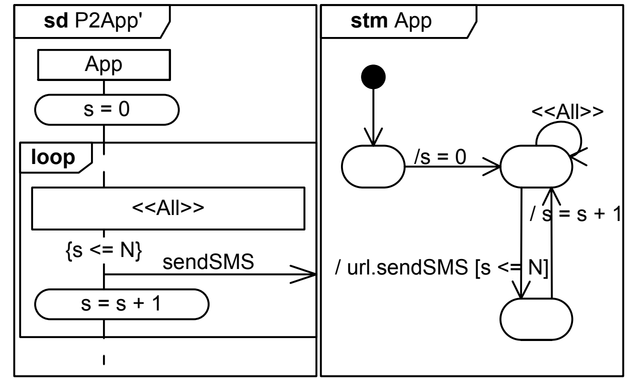

Figure 3. Sequence diagram to state machine

4 Step III: Transforming Sequence Diagrams to State Machines

In the third step of our method, the low-level sequence diagram is automatically transformed into a UML state machine which governs the behavior of an EM mechanism. The transformation has been in Java and Prolog.

We will use the diagram policy2 of Fig. 1 to illustrate the transformation process. First we convert policy2 into two diagrams; one diagram (which we call P2App) describing the lifeline App, and one diagram describing the lifeline url. Each of these diagrams must be converted into a basic UML state machine. In the following, we illustrate the transformation steps for the diagram P2App describing the lifeline App.

Before the diagram P2App is converted into a basic state machine, an intermediate pseudo-diagram P2App' is generated in which all inconclusive behavior has been made positive, and all negative behavior has been removed. Technically, the transformation first adds a pseudo construct All before each choice operator or event in the diagram, then all interactions that are encapsulated by the neg-operator are removed. The All construct is a placeholder all traces that do not contain events that are enabled directly after the occurrence of the construct.

The resulting diagram, named P2App' is shown on the left hand side of Fig. 3. Here the intermediate All construct has been inserted and the negative behavior of policy2 has been removed.

The goal now is to convert the sequence diagram P2App' into a basic state machine that describes exactly the positive traces of the sequence diagram. To achieve this, we make use of the operational semantics of sequence diagrams which has been shown to be sound and complete with respect to the denotational semantics of sequence diagrams as formalized in STAIRS [7].

The operational semantics defines a directed graph

whose nodes are diagrams, and whose edges are labeled by events, assignments, and so-called silent events which indicate which kind of operation has been executed. If the graph has an edge from a diagram d to a diagram d' that is labeled by, say, event e, then we understand that event e is enabled in diagram d, and that d' is obtained by removing e from d.

To transform the diagram P2App' into a state machine we use the operational semantics to construct the execution graph whose nodes are exactly those diagrams that can be reached from P2App' when executing the diagram without evaluating constraints or assignments. In this way we obtain the state machine shown on the right hand side of Fig. 3. There are three things to note w.r.t. the construction: (1) we have added an initial state which is not directly obtained from the execution graph, (2) all edges that contain silent events (which indicate the operator which has been executed) have been removed, and (3) the messages, assignments, and constraints have been converted to the corresponding state machine notation.

In Fig. 3, we have used the transition labeled All to represent transitions that are labeled by all messages in the alphabet of the state machine except for the message sendSMS. These transitions describe inconclusive behavior of the sequence diagram.

The final step is to parallel compose the basic state machines obtained for lifeline App and lifeline url. This results in a composite state machine. Semantically, an application is said to adhere to a state machine policy if each of its traces is described by the policy when all the events that are not in the alphabet of the policy are removed.

5 Conclusions and Related Work

We have presented a method for specifying high-level security policies that can be enforced by run-time monitoring mechanisms. We believe that the method is both easy to use and useful since it automates much of the policy formalization process.

Previous work that address the transformation of policies or security requirements are [1, 2, 3, 9]. All these works differ clearly from ours in that the policy specifications, transformations, and enforcement mechanisms are different from the ones considered in this paper.

The transformation of sequence diagrams (or a similar language) to state machines has been previously addressed in [4, 6, 13]. These works do not consider policies, nor do they offer a way of changing the granularity of interactions during transformation.

The only work that we are aware of that considers UML sequence diagrams for policy specification is [12]. The paper argues that sequence diagrams must be extended with customized expressions for deontic modalities to support policy specification. While this is true in general, this is not needed for the kind of prohibition policies that can be enforced by EM mechanisms.

Acknowledgement. The research on which this paper reports has partly been funded by the Research Council of Norway project SECURIS (152839/220) and partly by the EU through the S^{3}MS (contract no. 27004) project.

References

[1] Y. Bai and V. Varadharajan. On transformation of authorization policies. *Data and Knowledge Engineering*, 45(3):333–357, 2003.

[2] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006.

[3] M. Beigi, S. B. Calo, and D. C. Verma. Policy transformation techniques in policy-based systems management. *Proc. 5th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 13–22. IEEE CS Press, 2004.

[4] Y. Bontemps, P. Heymans, and P. Schobbens. From live sequence charts to state machines and back: A guided tour. *IEEE Trans. Softw. Eng.*, 31(12):999–1014, 2005.

[5] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS towards formal design with sequence diagrams. *Journal of Software and Systems Modeling*, 4(4):355–367, 2005.

[6] J. Klose and H. Wittke. An automata based interpretation of live sequence charts. *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–527. Springer, 2001.

[7] M. S. Lund and K. Stølen. A fully general operational semantics for UML 2.0 sequence diagrams with potential and mandatory choice. *Proc. 14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 380–395. Springer, 2006.

[8] R. K. Runde, Ø. Haugen, and K. Stølen. Refining UML interactions with underspecification and nondeterminism. *Nordic Journal of Computing*, 12(2):157–188, 2005.

[9] F. Satoh and Y. Yamaguchi. Generic security policy transformation framework for ws-security. *Proc. 2007 IEEE International Conference on Web Services*, pages 513–520. IEEE CS Press, 2007.

[10] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[11] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994.

[12] B. Solhaug, D. Elgesem, and K. Stølen. Specifying policies using UML sequence diagrams–an evaluation based on a case study. *Proc. 8th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 19–28. IEEE CS Press, 2007.

[13] J. Whittle and J. Schumann. Generating statechart designs from scenarios. *Proc. 22nd international conference on Software engineering*, pages 314–323. ACM, 2000.