# Engineering Self-protection for Autonomous Systems

Manuel Koch and Karl Pauls

Freie Universität Berlin,
Institut für Informatik,
Takustr. 9, D–14195 Berlin, Germany
{mkoch, pauls}@inf.fu-berlin.de

**Abstract.** Security violations occur in systems even if security design is carried out or security tools are deployed. Social engineering attacks, vulnerabilities that can not be captured in the relatively abstract design model (as buffer-overflows), or unclear security requirements are only some examples of such unpredictable or unexpected vulnerabilities. One of the aims of autonomous systems is to react to these unexpected events through the system itself. Subsequently, this goal demands further research about how such behavior can be designed and sufficiently supported throughout the software development process. We present an approach to engineer self-protection rules for autonomous systems that is integrated into a model-driven software engineering process and provides concepts to formally verify that a given intrusion response model satisfies certain security requirements.

## 1 Introduction

Model building as a means of producing appropriated documentation, providing specifications, and code generation is a standard practice in software engineering. Security, as an integral part of any modern software system that is not used in completely trusted environments, demands systematic support for software engineers who need to produce secure software. Considering security aspects throughout the entire software development process (and not during the requirements analysis and system integration phases only) by explicitly integrating security into the design models can aid in detecting and removing potential security breaches. Furthermore, model-centric and generative approaches, as the concept of Model Driven Architecture (MDA) [5], have led to advancing support for software engineers. Most noticeably, the *Model Driven Security* approach (see [3]) proves to not only define access-control languages (in this case *SecureUML*) but to provide a basis for refinement down to code as well. Access control is concerned with preventing unauthorized accesses to shared resources. Which accesses are authorized depends on specific security requirements and has to be specified in access control policies.

A model is an abstract part of the real world which contains the aspects relevant to the developer. It does not deal with any environment interaction. In particular from the viewpoint of security, not all possible attacks can be considered in a model. This may be because of the abstraction level of the model (e.g. buffer overflows cannot be seen in class diagrams) or because of environment parts that are not considered in a model (e.g. social engineering) or other reasons. Therefore, crucial functions must be monitored. Concerning security, accesses to crucial data must be logged to see if unauthorized access occur despite a given access control policy (e.g. by buffer overflows). In very sensitive environments, intrusion detection systems (IDS) [19] and security engineering [1], in practice, often need to be combined so that the occurrence of interactions and system states not covered by the normal use cases raises an alarm in an IDS. Consequently, the question of how to identify nonconforming or malicious behavior and how to react to discovered security breaches arises.

Ever since IBM's "call to arms" [6], called autonomous computing, research increasingly focuses more effort on self-adapting, self-protecting, and self-healing systems (i.e., autonoums systems) [18]. Monitoring the system is a necessary aspect in autonomous systems to detect system or environment changes and possibly to react on these changes if necessary. Self-repairing/Self-protecting systems are an ambitious goal whose realization concerns many aspects of software engineering (for example Baresi et. al considered self-healing in service-oriented systems in regard to dynamic binding of services in [2]). We consider in this paper the self-protection of the system in the case of successful security attacks.

We take up the challenge of providing sufficient support to design and realize self-protecting system. We present an approach to engineer self-protection rules for autonomous systems, integrated into a model-driven development approach, and capable to generate self-protecting access control aspects for XACML based infrastructures. Furthermore, we provide concepts based on graph transformations [17] to formally verify that a given intrusion response model satisfies certain security requirements.

The remainder of this article is organized as follows. We give next a description of the model-driven development approach and the underlying concepts of our access control model together with an operational semantic. Section 3 concerns the specification of protection rules. Section 4 presents the concepts to verify the satisfaction of security requirements. Finally, we present related work and conclusion.

## 2 Model-Driven Development

We first present the integration of our approach into a model-driven development approach namely, *Model Driven Security* and its underlying access control model *SecureUML*. It concerns the development of XACML [11] based access control policies and access control properties following an attribute based access control approach which can be described by a mapping of the modeling elements of *SecureUML* to XACML policies. Afterwards, an additional operational semantic of the entity operations is given that serves as a starting point for our self-protecting rules requirement analysis.

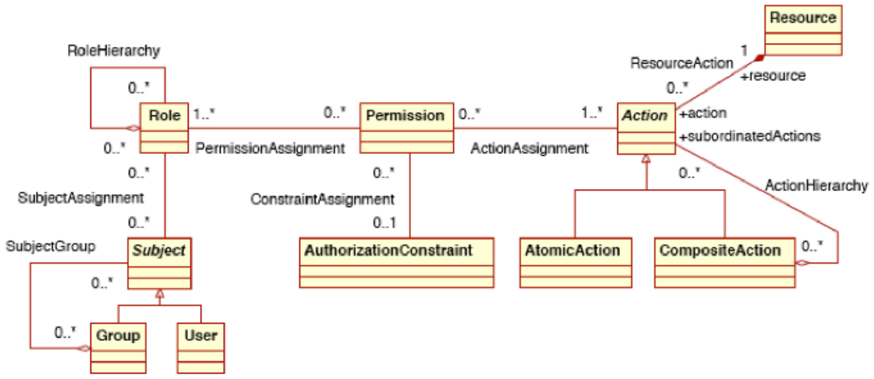


**Fig. 1.** RBAC metamodel

As already stated above, we build up on the *SecureUML* metamodel presented in [3]. Figure 1 presents the role based access control (RBAC) based metamodel that defines the abstract syntax for *SecureUML*. Due to space constraints we can not go in too much detail about the concrete syntax and semantics but refer the interested reader to [3]. On the left-hand side of the diagram RBAC is formalized. Users can be assigned to Groups. On the other side of the model, permissions, which can be assigned to roles, are used to model the ability to carry out actions on resources on behalf of a calling subject that is in a certain role. Authorization constrains can be used to constrain that certain permissions only hold in certain system states.

To this end, we require that subjects have or provide certain properties (credentials) to be assigned to roles. For example, a user name, a counter for logins, a printer quota, location in mobile scenarios etc. This approach is called attribute-based access control (ABAC) [16]. The main idea of ABAC is to dynamically define the authorization of subjects based on current property values of the calling subjects and their targeted resources, respectively. In addition to the relatively static defined roles, this attributes can be highly dynamic therefore, provide a way to capture the needs of e-commerce as well as enterprise and e-government applications in the internet ranging all the way to ubiquitous computing.

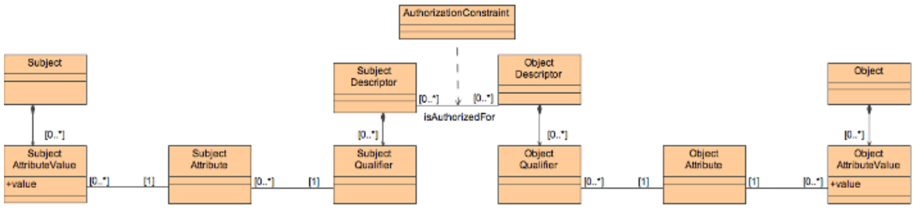


**Fig. 2.** ABAC model

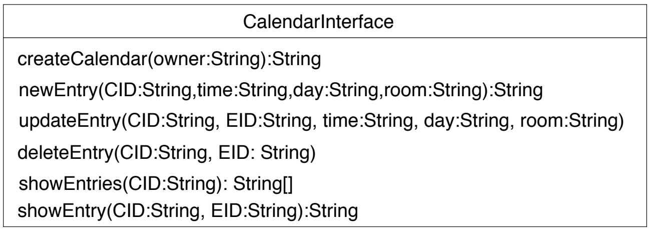
Figure 2 shows our base model of the applied ABAC approach. In general, we enhance the aforementioned metamodel in Fig. 1 by this explicit specification of subject and object attributes.

### 2.1 Generation of XACML Based Access Control Policies

*Model Driven Security* gives a means to integrate access control concerns into a model and subsequently generate code out of this model elements. In [3] concrete mappings for EJB and .Net are given. In addition, we focus on the generation of XACML based access control policies since, policy based infrastructures are more flexible and, more specific, are able to provide better support for policy changes and management then the standard security architectures of todays enterprise systems. To this end, we define and implemented a UML profile that enables to generate XACML policies from security models (i.e., models that are build using our ABAC enhanced *SecureUML* modeling elements).

### 2.2 Example and Operational Semantics of the Protection Model

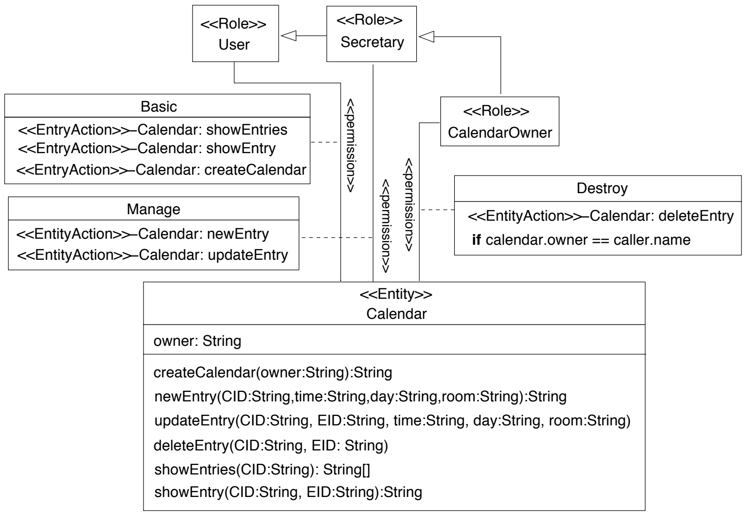
In addition to the SecureUML concrete semantic, our protection model contains an operation semantic of the entity operations. As a running example, we will consider developing a simplified version of a system for administrating calendars. Figure 3 shows the simple interface of the calendar application that basically, allows to create a calendar and subsequently, create, update, delete, and read entries (i.e., appointments).



```
CalendarInterface
createCalendar(owner:String):String
newEntry(CID:String,time:String,day:String,room:String):String
updateEntry(CID:String, EID:String, time:String, day:String, room:String)
deleteEntry(CID:String, EID: String)
showEntries(CID:String): String[]
showEntry(CID:String, EID:String):String
```

**Fig. 3.** Interface of the Calendar Application

Considering this simple interface, one may want to enforce some basic integrity properties like, for example that the creator of a calendar becomes its owner hence, is the only one that is allowed to delete entries. Arbitrary users are allowed to read the entries of any calendar but modification is up to the owner of the calendar or a substitute like a secretary. At the very least a secretary should be able to make and manage appointments (i.e., create and update entries in the calendar). Lets assume that the deletion of an entry is restricted to the owner of the calendar only. These security requirements are implemented in the model in Fig. 4 which is an instance of the *SecureUML* metamodel. We have three roles



<<Role>> User
<<Role>> Secretary
<<Role>> CalendarOwner
Basic
<<EntryAction>>–Calendar: showEntries
<<EntryAction>>–Calendar: showEntry
<<EntryAction>>–Calendar: createCalendar
Manage
<<EntityAction>>–Calendar: newEntry
<<EntityAction>>–Calendar: updateEntry
Destroy
<<EntityAction>>–Calendar: deleteEntry
if calendar.owner == caller.name
<<permission>>
<<permission>>
<<permission>>
<<Entity>> Calendar
owner: String
createCalendar(owner:String):String
newEntry(CID:String,time:String,day:String,room:String):String
updateEntry(CID:String, EID:String, time:String, day:String, room:String)
deleteEntry(CID:String, EID: String)
showEntries(CID:String): String[]
showEntry(CID:String, EID:String):String

**Fig. 4.** security model

and three permissions. The permission *Basic* allows a subject in role User to read calendar entries and to create calendars. The permission *Manage* allows subjects in role Secretary to modify a calendar and permission *Destroy* allows subjects in role CalendarOwner to delete calendar entries.

Once the security model is accomplished, the operational semantics of the entity operations have to be specified. We define the notion of a *protection model* as follows. A *protection model* is a pair of a security model (as described above) and a set of transformation rules [17]. Considering our example we write, the security model of figure 4 as $M$ and the protection model as a pair $(M, ORules)$. Figure 5 gives the transformation rules for the creation of a calendar and the deletion of entries in the calendar via the $create()$ and $deleteEntry()$ methods of the *Calendar* entity. A transformation rule consists of two object diagrams. The diagram on the left-hand side of a rule models the precondition to apply the rule. The object diagram on the right-hand side models the transformed object state. The left-hand side of rule *create(x)* requires a subject with name $x$ in a role and this role must have a permission with entity action create on the calendar entity. If this object structure can be found in a system state, a new calendar object for the subject with name $x$ is created. The left-hand side of rule *deleteEntry* requires a subject and a connected calendar object. The subject must be in a role which has a permission for entity action *deleteEntry*. The condition $calendar.owner == x$ enforces that the rule can be only applied if the subject is the owner of the calendar. The effect of the rule is the deletion of an entry of the calendar object. Transformation rules can be mapped to graph rules to give them a formal semantics [8, 17].

Ultimately, the transformation rules capture the aforementioned attributes of our subject and object descriptors and more importantly, their changes during the state changes of the system. Since we assume that state changes are triggered

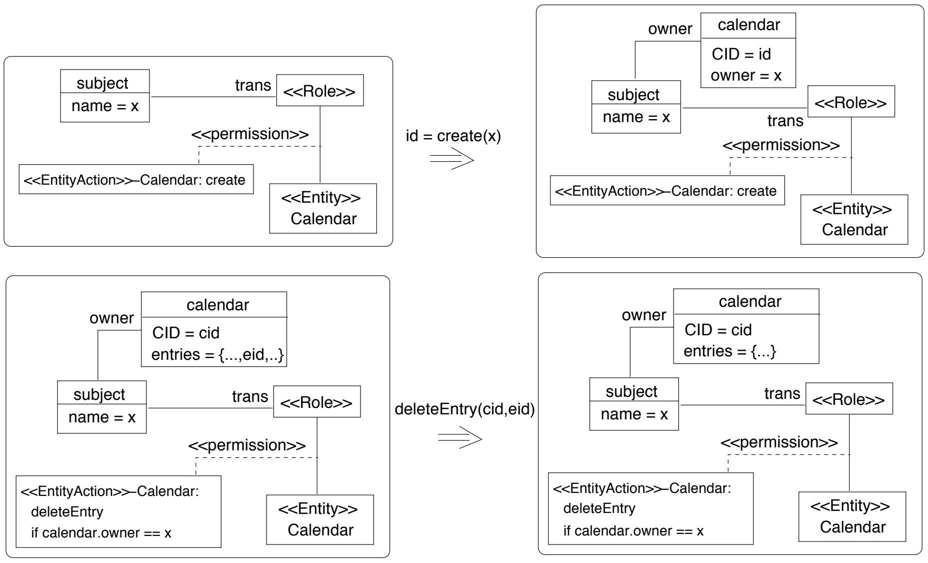


**Fig. 5.** Operational semantics for calendar operations create() and deleteEntry()

by method calls our notion is sufficient to model their impact on the protection model. In figure 5 for example, the create(x) call to the calendar interface sets the owner attribute of the calendar to the name of the caller. In consequence, we can build up on this attribute to restrict the deletion of an entry to the actual creator of the calendar (i.e., its owner) by comparing the name attribute of the caller with the owner attribute of a targeted calendar.

## 3 Specification of Self-protection Rules

The analysis of security requirements for a software system is a difficult design task and recent research focuses on developing models and concepts to elicit, analyze and document security requirements [9, 10]. We assume in this article, that security requirements are documented and a risk assessment has given them a priority. The following list shows the examples used in the remainder of this article.

*Security Requirement C1:* Prevent that a calendar has more than one owner.

*Security Requirement C2:* Prevent that a user is logged into the system more than once.

*Security Requirement C3:* Prevent denial of service attacks by creating more than $n$ calendars.

These in natural language formulated requirements can be specified in semi-formal or formal constraint languages (e.g., OCL[13]) and models can be checked if they satisfy these requirements. The focus of this article, however, is not this static check of design models, but we are interested in the security vulnerabilities that are detected during run-time even if the model is previously checked to be secure.

Therefore, a crucial component of autonomous systems is a monitor which observes the system states during run-time in order to detect constraint violations. When the monitor detects an insecure system state the system should react by protection means (in the case of an autonomous system the system reacts autonomously). One possible solution to protect the system would be a system shutdown or to disconnect the whole system. This solution, however, is quite rigid and restricts the availability of the system often more as necessary. If the originator of a security violation can be determined it would possibly be enough to eliminate this user from the system. If the originator is unknown, it is sufficient to disconnect the attacked subsystem or restrict its functionality, so that the remaining system can continue working in a restricted mode.

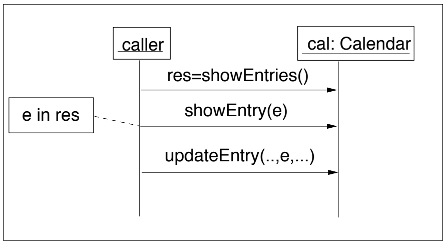
Before we present an approach to specify a more fine-grained protection, we differentiate between *self-protection* and *self-repairing* of the system. Self-protection changes the security model by revoking permissions as far as necessary so that an intruder cannot do any harm with the acquired authorization, but the system state remains unchanged. Self-repairing, on the other hand, transforms the insecure system state into a secure state and lets the security model unchanged. Self-repairing, i.e. an automatic modification of the system state without any interaction with an administrator, is often difficult to implement. Consider as an example a violation of the requirement $C1$ from above, i.e., the monitor detects two calendar owners for a calendar. Should we revoke both owners from the calendar (but then we have calendars without owners) or should we only revoke one calendar owner (but which one, which owner is the "real" owner)? In the case of a violation of requirement $C3$, calendars must be removed to reach the maximum boundary of allowed calendars. But, which calendar should be removed?

We focus next on the specification of self-protection. For each security requirement, a set of protection rules models the reaction of the system to the violation of the security requirement and transforms the protection model. The transformation should restrict the model as far as necessary and should allow system availability as far as possible. The protection rules are developed in two steps:

1. *Specify the response requirement.* A response requirement for a security requirement specifies the system functionality which must be restricted in the case of a security requirement violation.
2. *Specify the protection sets for the response requirement.* A protection set contains a set of transformation rules to restrict the security model. The rules of a protection set for a response requirement shall satisfy the response requirement.

## 3.1 Development of the Intrusion Response

To support the system designer in finding the appropriate response requirements, we suggest an approach which is driven by the UML models, since static diagrams (as class diagrams) contain the elements that should be protected (in our example the calendars and their entries), the behavior diagrams (as sequence diagrams) show how the protected elements are accessed. Therefore, the designer decides on the basis of these UML models the measures to do in the violation response. Consider the security requirement $C1$ for at most one calendar owner as an example. The designer has the class diagram in Fig. 4 and the sequence diagrams in Fig. 6 as documentation and assumes now that requirement $C1$ can be violated by a security vulnerability so that an attacker can become owner of calendars of other persons. When the designer considers the sequence diagrams



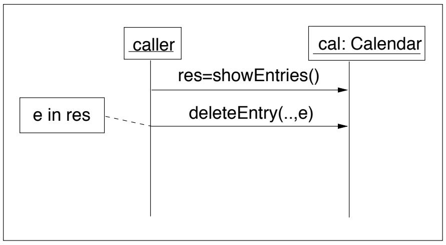


**Fig. 6.** Sequence diagrams for updating a calendar entry and for deleting an entry

in Fig. 6, (s)he realizes that the attacker can call the operations *showEntries()*, *showEntry()*, *updateEntry()* and *deleteEntry()*. While an unauthorized call of the operations *showEntries()* and *showEntry()* appears to be an acceptable (i.e., it does not concern integrity) risk, compared to disabling read access for all (including the trustworthy) users, an unauthorized call of the operations *updateEntry()* and *deleteEntry()* cannot be tolerated. Therefore, the designer adds *updateEntry()* and *deleteEntry()* for calendar owner to the response requirement, i.e., both operations must not be called by calendar owners when requirement $C1$ is violated. Analog, the response requirements for the other security requirements are specified driven by the UML diagrams.

The list below shows the response requirements of our calendar example. A response requirement is a set of pairs $(Caller, Operations)$ consisting of a list of callers $Caller$ who are not allowed to call the operations in the set $Operations$ in the case of the corresponding security violation.

$$Response(C1)=\{(CalendarOwner,\{deleteEntry(), newEntry(), updateEnty()\})\}$$
$$Response(C2)=\{(User,\{deleteEntry(), newEntry(), updateEnty(), createCalendar()\})\}$$
$$Response(C3)=\{(User,\{createCalendar()\})\}$$

### 3.2 Specification of Self-protection

A first idea to satisfy the response requirement would be to generally disallow callers to call the operations in the response requirement. Since the response requirements are connected to certain callers, however, this general prohibition is too strong. To restrict the operations to certain callers requires additional operation conditions. Since operations are implemented and a code change during run-time is not desirable, operations cannot be modified with respect to the response requirement. Therefore, the security model must be modified so that only the specific callers are affected. A change of the security model can be done during run-time and is immediately enforced by the XACML infrastructure [11].

The security model transformation is specified by a set of graph rules (Fig. 7 shows a part of the graph rules for the calendar example). The *protection sets* for a response requirement contain a subset of the protection rules. The protection set $Protect(C1)$ to satisfy the response requirement $Response(C1)$ removes all permissions from role calendar owner and adds a permission to calendar owner to read calendars. The protection set $Protect(C2)$ removes all permissions to modify a calendar and introduces a restricted basic permission which allows the user to read the calendars only. The protection set $Protect(C3)$ removes the permission to create calendars by adding a restricted basic view.

$Protect(C1)$={remove destroy(CalendarOwner), remove inheritance, add basic(CalendarOwner)}.
$Protect(C2)$={remove destroy(CalendarOwner), remove manage(Sectretary), replace basic, add basicrestricted(User)}.
$Protect(C3)$={replace basic, add basicrestricted(User)}.

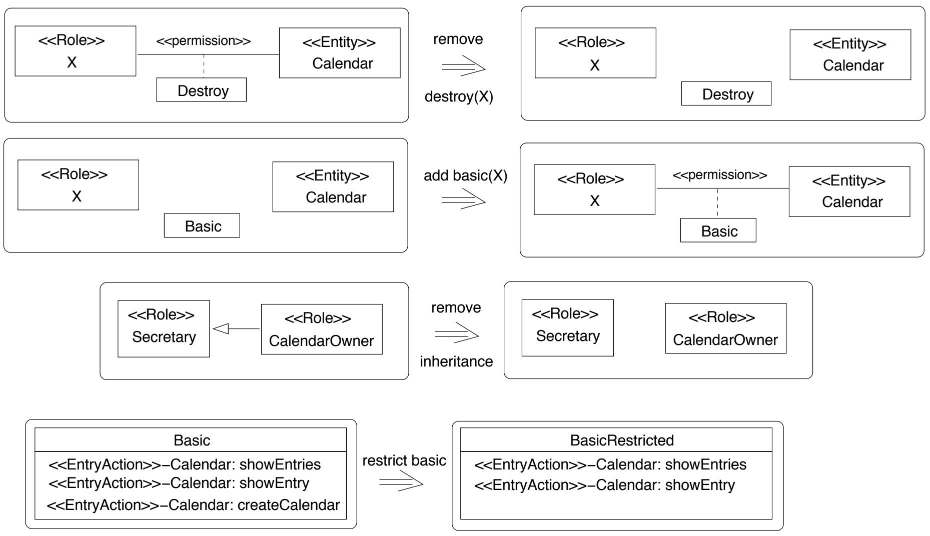


**Fig. 7.** The protection rules

When the system monitor detects a violation of a security requirement (e.g., $C1$) the rules in the protection sets are executed (e.g., $\mathit{Protect}(C1)$) to ensure the response requirement (e.g., $\mathit{Response}(C1)$). Figure 8 shows the results of applying the protection sets $\mathit{Protect}(C1)$, $\mathit{Protect}(C2)$ and $\mathit{Protect}(C3)$, respectively, to the security model in Fig. 4. When several security requirements

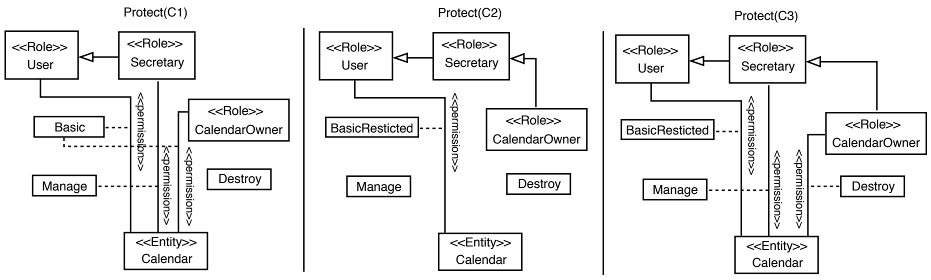


**Fig. 8.** The security models after execution of $\mathit{Protect}(C1)$, $\mathit{Protect}(C2)$ and $\mathit{Protect}(C3)$

are violated at the same time, several protection sets are applied. The response requirement for two security requirements $C1$ and $C2$ is $\mathit{Response}(C1 + C2) = \mathit{Response}(C1) \cup \mathit{Response}(C2)$. We define the protection set as $\mathit{Protect}(C1 + C2) = \mathit{Protect}(C1) \cup \mathit{Protect}(C2)$.

One could argue that, instead of specifying the protection rules, it would be easier to specify immediately the restricted security models. Since there must be a security model for each combination of violated security requirements, one has to specify $2^n - 1$ security models in the case of $n$ requirements. Therefore, for a bigger $n$ it is certainly more convenient to specify $n$ rule sets which ensure that each constructed security model is consistent. The next section concerns this consistence statement.

## 4 Protection Satisfaction

A protection set contains rules which modify the security model in the case of unexpected security requirement violations. By now, there is no restriction on the ordering in which the rules of a protection set must be applied and one can wonder if any order results in the same security model or if the ordering is relevant. A second question is whether the security model constructed by the rules of a protection set satisfy the response requirement. Therefore, this section concerns the following questions.

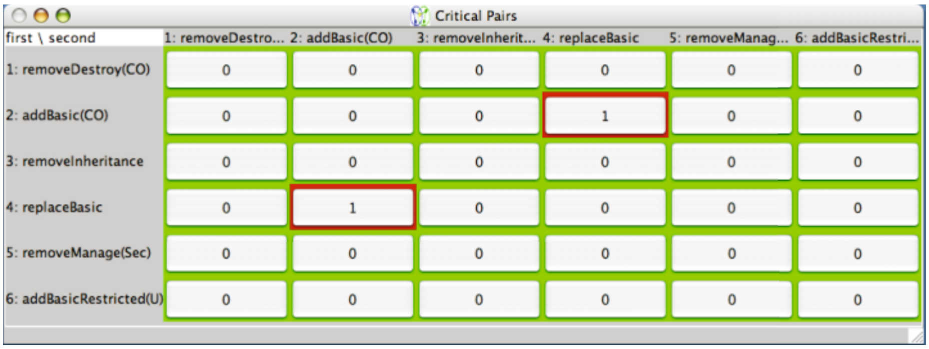
1. Does the rule application ordering influence the final security model?
2. Does a protection set satisfies a response requirement?

## 4.1 Dependencies Between Protection Rules

If a protection set becomes necessary to protect the system against a security violation, each rule in the protection set is applied once. Since the ordering is by default unrestricted, the following problems may occur (see also independence of graph transformations in [17]).

*Problem 1:* Assume rules $p_1$ and $p_2$ which are both applicable to the security model $M$, but rule $p_1$ deletes elements required by $p_2$, so that an application of $p_1$ prevents the applicability of $p_2$. Dependent on the rule ordering, two different security models $M'$ are generated and, therefore, the two rules are in conflict. To detect these conflicts, *critical pair analysis* of graph rules [15, 4] can be used. The critical pairs for two rules are constructed by overlapping the rule left-hand sides in all possible ways, such that the intersection contains at least one deleted element. In this way, critical pairs show all the potential conflicts between the rules in a minimal context. Each actual conflict in a bigger context will be represented by one of the critical pairs.

There is tool support for generating the critical pairs for rules implemented in the AGG tool [20]. Figure 9 shows the result of the critical pair analysis for our example rules. The tool detects a critical pair for rules *replaceBasic* and *addBasic(CO)*. This bases on the fact, that rule *replaceBasic* deletes the permission *Basic* which in turn is required in the left-hand side of rule *addBasic(CO)*. Therefore, applying first rule *replaceBasic* prevents the application of rule *addBasic(CO)*.


Critical Pairs

| first \ second | 1: removeDestro... | 2: addBasic(CO) | 3: removeInherit... | 4: replaceBasic | 5: removeManag... | 6: addBasicRestri... |
|---|---|---|---|---|---|---|
| 1: removeDestroy(CO) | 0 | 0 | 0 | 0 | 0 | 0 |
| 2: addBasic(CO) | 0 | 0 | 0 | 1 | 0 | 0 |
| 3: removeInheritance | 0 | 0 | 0 | 0 | 0 | 0 |
| 4: replaceBasic | 0 | 1 | 0 | 0 | 0 | 0 |
| 5: removeManage(Sec) | 0 | 0 | 0 | 0 | 0 | 0 |
| 6: addBasicRestricted(U) | 0 | 0 | 0 | 0 | 0 | 0 |

**Fig. 9.** Computation of critical pairs by AGG

After the computation of critical pairs for a protection set, the rule set can be divided into conflict free rules (rules which do not have critical pairs) and conflicting rules. Conflict free rules can be applied in any order (from the viewpoint of problem 1, we will see next another problem which additionally influences the rule application order) and the security engineer can use them in any combination in the protection sets to get a single final security model.

Conflicting rules should not be used together in a protection set or the security engineer must specify the desired application order. In our example, (s)he could specify that *addBasic(CO)* must always be applied before *replaceBasic*.

*Problem 2:* Assume now rules $p_1$ and $p_2$, so that $p_1$ creates elements required by $p_2$. Then, rule $p_2$ can be applied only after $p_1$, but not before (see also sequential dependence in [17]). These conflicts are called *sequential dependence* conflicts and can be detected by considering the overlaps of the right-hand side of rule $p_1$ and the left-hand side of rule $p_2$. There is no conflict if the left-hand side of $p_2$ does not require elements which are generated by $p_1$. Otherwise, there is a conflict. If we investigate the rules of our example, we see that the rule *addBasicRestricted(U)* is sequential dependent of *replaceBasic* since *replaceBasic* generates the permission *BasicRestricted* required by rule *addBasicRestricted(U)*. All other rules are not sequential dependent.

Analog to critical pair analysis, sequential dependencies between protection rules can be automatically detected and presented to the security engineer who uses this information in the specification of the protection sets.

## 4.2 Satisfaction of the Response Requirement

Applying a protection set to a protection model $(M, ORules)$ results in a new protection model $(M', ORules)$ in which the security model is changed (from $M$ to $M'$). The operation rules $ORules$ remain unchanged under this transformation. The permission or denial of operation accesses must now be checked with respect to the new security model $M'$.

A protection set *satisfies* a response requirement $Res$, if for any pair $(Caller, Operations)$ in $Res$, none of the transformation rules for an operation in $Operations$ can be applied to $Caller$ in the changed security model $M'$. This satisfaction can be checked by considering the left-hand sides of the transformation rules in the response requirement $Res$ and the new model $M'$. If the security relevant part of the left-hand side (which consists of all elements with stereotype <<Role>>, <<EntityAction>>, <<Permission>> and <<Entity>>) of a rule $p$ in $Res$ can be embedded into the security model $M'$ then one can construct a state for $M'$ to which $p$ can be applied (mainly the left-hand side itself). Therefore, the response requirement is not satisfied. On the other hand, if the security relevant part of the rule cannot be embedded into the security model, this part can neither be embedded into a state for $M'$. This means that the rule is never applicable and the response requirement is satisfied.

Consider as an example the protection set $Protect(C1)$ for the security requirement $C1$. The modified security model $M'$ is shown in Fig. 8 on the left-hand side. The response requirement $Response(C1)$ forbids a calendar owner for example to call the operation *deleteEntry()*. The security relevant part of the left-hand side of the transformation rule for *deleteEntry()* (bottom of Fig. 5) cannot be embedded into the security model $M'$, since the rule requires a role which has a permission on the calendar entity, and the permission contains an entity action *deleteEntry*. In the security model in Fig. 8, however, no role is connected to the permission *Destroy* (the only permission with entity action *deleteEntry*). Therefore, the rule for *deleteEntry()* cannot be applied to any system state corresponding to the security model $M'$.

## 4.3 Benefits for the Security Engineer

At the end of this section, we summarize how the answers of the two questions in the beginning of this section can support the security engineer in designing a self-protection system.

1. *Does the rule application ordering influence the final security model?*
   Paragraph 4.1 has shown that different rule application orderings may lead to different security models. The security engineer, however, can use critical pair analysis (supported by the AGG tool) and sequential dependence analysis to compute the conflicting rules. Considering these results in the engineering process of the protection sets allows the security engineer to get a deterministic behavior of the protection set response.
2. *Does a protection set satisfies a response requirement?*
   Paragraph 4.2 has presented a way to check whether the rules in a protection set satisfy a response requirement by considering the left-hand side of the transformation rules which specify the operations in the response requirement. If the designer detects rules in a protection set which does not satisfy a response requirement, (s)he must change the protection set or the protection rules until all response requirements are satisfied.

## 5 Related Work

Our approach uses the security engineering model presented in [3] for which tool support is given by an integration of the SecureUML metamodel into the ArcStyler tool [12]. The analysis stage of the software process, however, is not considered but the process starts with the design models. Jürjens presents in [7] the integration of security into the UML. He shows how to model several security aspects by UML model elements as, for example, stereotypes or tagged values. His approach is more general than ours since it is not restricted to access control but considers, for example, also security protocols. In [14, 21] approaches to design intrusion detection systems are presented. The design, however, focusses on the detection of attackers, less on the design of the response of an attack. Baresi et. al considered self-healing in service-oriented systems in regard to dynamic binding of services in [2].

## 6 Conclusion and Future Work

We presented a model-driven approach to engineer self-protection for autonomous systems. The approach is integrated into model driven security *SecureUML* for modeling access control and supports the system designer in engineering self-protection rules to react to unexpected security vulnerabilities.

Self-protection is specified by a set of transformation rules which restrict the security model. A graph-based semantics for the transformation rules allows us to verify that security requirements are satisfied by the specified self-protection rules.

We target an XACML based infrastructure which enforces the security model transformation that result by the self-protection sets. Furthermore, the XACML policies shall be generated from the models and protection rules. Another point of future work is the specification of the cancelation of self-protection restrictions. In other words, if the reason that causes the insecure state is eliminated, we have rules which transform the restricted model back into an unrestricted safe system.

## References

1. G.-J. Ahn and R. Sandhu. Role-Based Authorization Constraints Specification. *ACM Transactions on Information and System Security*, 3(4):207–226, Nov. 200.
2. L. Baresi, C. Ghezzi, and S. Guinea. Towards Self-healing Compositions of Services. In *Proc. of PRISE'04, First Conference on PRInciples of Software Engineering*, pages 11–20, 2004.
3. D. Basin, J. Doser, and T. Lodderstedt. Model Driven Security: from UML Models to Access Control Infrastructures. *Journal of ACM Transactions on Software Engineering and Methodology*, 2005.
4. H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conf. on Graph Transformation*, number 3256 in LNCS, pages 161–177. Springer, 2004.
5. D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* John Wiley and Sons., 2003.
6. P. Horn. Autonomic computing: IBM perspective on the state of information technology. Technical report, IBM T.J. Watson Labs, October 2001.
7. Jan Jürjens. *Secure Systems Development with UML.* Springer, 2005.
8. M. Koch and F. Parisi-Presicce. Access Control Policy Specification in UML. In *Proc. of UML2002 Workshop on Critical Systems Development with UML*, number TUM-I0208, pages 63–78. Technical University of Munich, September 2002.
9. M. Koch and K. Pauls. Generation of Role-based Access Control Requirements from UML Diagrams. In *Proc. of SREIS 2005, Symposium on Requirements Engineering for Information Security*, 2005.
10. N.R. Mead and T. Stehney. Security Quality Requirements Engineering (SQUARE) Methodology). In *Proc. of Software Engineering for Secure Systems (SESS05)*, 2005.
11. OASIS. *XACML 1.1 Specification*, August 2003.
12. Interactive Objects. Arcstyler, 2005. `www.io-software.com`.
13. OMG. *OCL 2.0 Specification, Version 2.0.* OMG, 2005.
14. M. M. Pillai, Jan H. P. Eloff, and H. S. Venter. An approach to implement a network intrusion detection system using genetic algorithms. In *SAICSIT '04: Proceedings of the 2004 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 221–221, , Republic of South Africa, 2004. South African Institute for Computer Scientists and Information Technologists.

15. D. Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In M. Sleep, M. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting*, pages 201–214. Wiley, 1993.
16. T. Priebe, W. Dobmeier, B. Muschall, and G. Pernul. ABAC – Ein Referenzmodell für attributbasierte Zugriffskontrolle. In *Proc. of Sicherheit 2005*, pages 285–296. Lecture Notes in Informatics GI–Edition, 2005.
17. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
18. R. Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering - A NASA Journal*, 1(1), 2005.
19. M. Stillerman, C. Marceau, and M. Stillman. Intrusion Detection for Distributed Applications. *Communications of the ACM*, 42(7):62–69, July 1999.
20. G. Taentzer, C. Ermel, and M. Rudolf. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter The AGG Approach: Language and Tool Environment. World Scientific, 1999.
21. Giovanni Vigna, Fredrik Valeur, and Richard A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–97, New York, NY, USA, 2003. ACM Press.