Automated Checking of SAP Security Permissions

Sebastian Höhn[⋆] and Jan Jürjens

Software & Systems Engineering, Informatics, TU Munich, Germany

Abstract. Configuring user security permissions in standard business applications (such as SAP systems) is difficult and error-prone. There are many examples of wrongly configured systems that are open to misuse by unauthorised parties.
To check permission files of a realistic size in a medium to large organisation manually – a typical number would be 60,000 entries – can be a daunting task which is often neglected.
We present research on construction of a tool which automatically checks the SAP configuration for security policy rules (such as separation of duty). The tool uses advanced methods of automated software engineering: The permissions are given as input in an XML format through an interface from the SAP system, the rules are formulated as UML specifications in a standard UML CASE tool and output as XMI, and our tool checks the permissions against the rules using an analyser written in Prolog. Because of its modular architecture and its standardised interfaces, the tool can be easily adapted to check security constraints in other kinds of application software (such as firewall or other access control configurations).

1 Introduction

The management and configuration of security-related resources in standard business applications is one of the most important tasks in mission-critical departments. There is not only the potential of a negative impact of public disclosure of confidential information and a resulting loss of faith among customers, but the threat of direct financial losses. Computer breaches are a real threat as a study by the Computer Security Institute shows:

- Ninety percent of the respondents detected computer security breaches within the last twelve months.
- Forty-four percent of them were willing and/or able to quantify their losses. These 223 firms reported $455,848,000 in financial losses [Pow02].

This examples shows the vast impacts security breaches can have. One of the most neglected facts are incidents caused by employees. The cited study of the CSI shows that for the respondents that had 1 to 5 incidents, 42 % of these incidents were caused from within the company [Pow02]. This demonstrates the importance of properly configuring security permissions in business applications.

It is important to realise that the existence of security mechanisms itself does not provide any level of security unless they are properly configured. That this is actually the case is often non-trivial to see. This applies especially to the financial sector, where the permissions have to satisfy more complex correctness conditions. One example is the rule of "separation-of-duty", meaning that a certain transaction should only be performed jointly among two distinct employees (for example, granting a large loan). Difficulties arise firstly from the inherent dynamics of permission assignment in real-life applications, for example due to temporary delegation of permissions (for example to vacation substitutes). Secondly, they

[⋆] Contact: `hoehn@in.tum.de` . Boltzmannstr. 3, 85748 München/Garching.

arise from the sheer size of data that has to be analysed (in the case of the large German bank, for whom the current work is supposed to be performed, some 60,000 data entries). A manual analysis of the security-critical configurations through system administrators on a daily basis is thus practically impossible, resulting in gaping security holes in practise. This observation motivated the current research which has been initialised in a cooperation with a large German bank and their security consulting partner. The goal was to develop a tool which can be used to automatically check security permissions against given rules in a specific application context (such as the separation of duty rule in the banking sector). The tool should in particular be applied to analyse the SAP security permissions of the bank at hand. The current paper reports on the design and development of this tool.

We present research on the design and construction of a tool which automatically checks the SAP configuration for security policy rules (such as separation of duty). The permissions are given as input in an XML format through an interface from the SAP system, the rules are formulated as UML specifications in a standard UML CASE tool and output as XMI, and our tool checks the permissions against the rules using an analyser written in Prolog. Because of its modular architecture and its standardised interfaces, the tool can be easily adapted to check security constraints in other kinds of application software (such as firewall or other access control configurations).

In the next section, we explain the task that the tool is supposed to solve in more detail (including the format of permissions and rules to be supported), as well as the architecture of and the underlying concepts and important design decisions regarding the tool. Section 3 explains the actual analysis performed in the tool at the hand of some examples. We close with a discussion of related work and a conclusion.

2 Automated Analysis of Security Rules

2.1 The Goals

As explained above, the correct configuration of secure business applications is a challenging task. So there is a need for automated tool-support. The tool presented here takes a detailed description of the relevant data structure of the business application, the business data, and some rules written by the administrator. Using this information, the tool checks whether the rules hold for the given configuration. If the rules do not hold this is written to the generated security-report. The tool should be able to accomplish the following specific tasks:

- It should read the configuration from the business application.
- It should automatically generate a report of possible weaknesses.
- It should provide a flexible configuration of the report's data.
- It should be easily configurable for different business applications.
- It should be able to check large-scale databases.
- The checking should be based on freely configurable rules.

Two other goals are particularly important to enable use of the tool beyond the specific task of checking SAP permissions of the SAP installation at hand: it has to be easy to integrate the tool with different business applications, and the rules that have to be checked need to be very flexible.

To make the tool as flexible as possible and on the other hand as easy to use as one could, a modular design is of great importance. Also, it is important to use well-known techniques for describing the information necessary for the tool, to facilitate use in practise by ease of learning how to use the tool. This motivated the use of the widely used Unified Modelling Language (UML) to formulate this information.

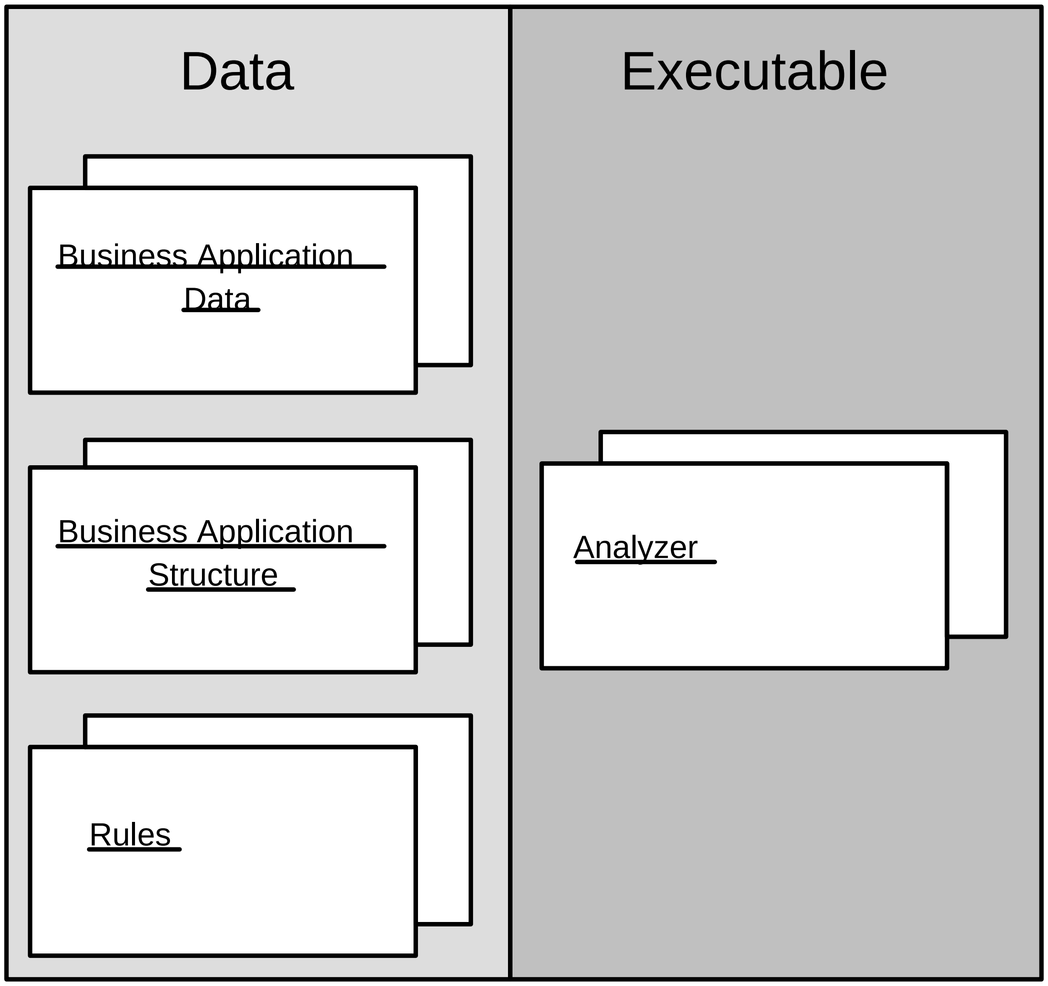

Fig. 1. Overview of the Tool's Architecture

2.2 Architecture

The tool mainly consists of three parts. They store the information describing the relevant data structure of the business application, define the rules and evaluate the rules. An additional part is needed to import the data from the business application (such as the SAP system). As in our example this is the user data and some structural information about transactions.

The complete separation of the tool and the business application provides additional security and privacy: Firstly, by separating the tool from the business application, there is no way the tool could add any weaknesses to this security-critical part of the company's IT-system. Secondly, this way it can be made sure that only the information needed for the analysis is exported to a foreign tool, which is important privacy matters. Both aspects should facilitate adoption of the tool.

The information itself is completely stored in XML. The business application's data has to be exported to XML files. In the specific application of the tool to SAP security permissions, this task is performed by one of the project partners and outside the scope of the current paper.

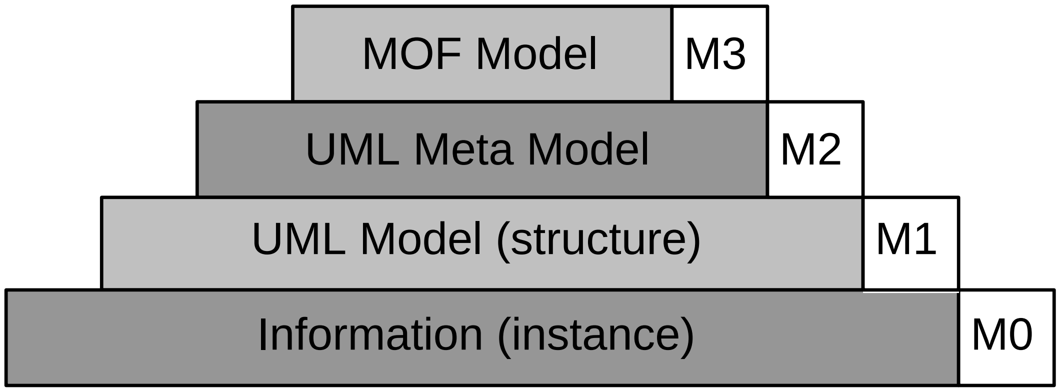

Fig. 2. Meta-model Framework according to OMG

The data structure of the business application is defined by UML class diagrams. Any case tool capable of saving XMI data can thus be used to do the modelling. Rules are stored in XML as well. There is a graphical user interface in development which will help with the creation of rules, but this extension is not really required for using the tool: any input of rules can be added to the tool as long as it creates XML valid for the tool's schema files. With the information above, the tool can check the rules and create the report.

As an option, the report can use templates to generate the layout that the user wants. To adapt the level of information to the given needs, every rule has a "level of verbosity". Then the rule is only evaluated if the report's desired "level of verbosity" is higher than the rule's level. So it is possible to adapt the report to the needs of several situations.

2.3 The Business Application as a Model

Following conventions published by the Object Management Group (OMG) as "the classical four layer meta-model framework" [Obj02], software systems can be modelled particularly flexibly in an approach based on several layers of information (see Figure 2). Throughout the description of the analyser there will be several types of information that fit into different layers on OMG's meta-model framework. In this framework there are UML models on layer 1 (M1) and application data on layer 0 (M0) (see [Obj02] pp. 2-2 to 2-3).

According to this separation of "model" and "information" the analyser needs two distinct types of data. First it needs "meta data" which is the description of the data structure of the business application itself and is given as an UML model of the application. This is what sometimes is called the "structure of the business" application and it is on level M1. On the other hand the analyser needs to know about the data itself, this is what is called "instance data" and it is information on level M0.

To illustrate the separation of data on layer M1 and data on layer M0 we consider an example. Assume there is "some" user-data in the business application. Every user has a name and a password. To formally describe the meaning of "some" in the expression "some user-data" there is a "model" that tells the tool about the class user and it's attributes name and password. This is done with an UML model and is data on level M1. When the tool checks the rules and needs to evaluate information of some special user e.g. "John", it needs what is called "information" in the "meta-model framework". This information is called "instance-data" and it is given as XML documents (this is, as the analyser uses it, placed on layer M0) [Obj02].

2.4 Permissions

To associate permissions for transactions via roles to users in role based access control (RBAC), the tool uses UML class diagrams. These diagrams can be directly used to give this information, and we do not need to introduce any additional features. The tool reads the class diagram and evaluates classes and associations.

In general, the analyser is not restricted to such an RBAC model or to any specific model at all. It is capable of evaluating rules on any class diagram that has the connection attributes assigned as names of the associations and the direction of associations defined by the navigable flag. The analyser evaluates the model as a graph with classes as nodes and associations as edges, where edges are directed. As we will see later, for the evaluation of rules, we need to require that there must be a path between the two classes involved in that rule, and there must be instance data so that the connecting attributes of each class match.

To explain this in more detail, we consider the example in Figure 3: the class diagram assigning permissions to users consists of the classes user, role, transaction, and permission, with attributes as in Figure 3. There is an association role_id between user and role, an association role_id between role and transaction, and an association transaction_id between transaction and permission. The analyser uses this model to automatically find a user's permissions.

Note that when assigning a permission p to a user u via a role r, and the user u also happens to have another role r', then (of course) it is not admissible to conclude that any user u' with the role r' should also be granted the permission p. In that sense, assigning permissions to users via roles is "uni-directional". In the class diagrams defining permissions, this is specified by using the "navigable" flag of UML class diagrams. This flag is an attribute of an association's endpoint. If this flag is set to "true" at the endpoint of a class c (signified by an arrow at that side of the association), our rule-analyser may associate information from the other end of the association with c. If it is set to "false", this information may not be evaluated. This way our tool may gather the permissions with respect to transactions granted to a given user by traversing the class diagram along the associations in the navigable directions permitting a "flow of information". This way the tool "collects" all users that have a given role, but does not recursively collect all users that have any of the roles that a given user has (as explained above).

To know how the elements in the application are connected there must be some kind of ID that can be evaluated at both sides of the connection. As in the short example above the user would have some kind of "role-id" in his user data, and a role would have the same id. The application retrieves the user's "role-id" and finds the role with the same id. To express that mechanism in our static UML class diagram, there is an association between the classes that exchange information. The user-class would be associated with the role-class, so the tool knows there is some kind of interaction (i.e., the application is able to find a user's role). To enable the search of information the analyser implicitly adds an additional attribute to either class at the association's endpoints. These new attributes are assigned the association's name.

As with every programming environment, class names have to be unique throughout the data structure of the business application. That should not cause a restriction in any environment.

2.5 Instance Data

Besides the structural data elements explained above, we need so-called "instance data". Here an instance may, for example, be a real user of the system. This information is very important for most of the rules one would like to evaluate. There are, of course, rules that do not need instance data (if one is checking the UML data structure model itself for some

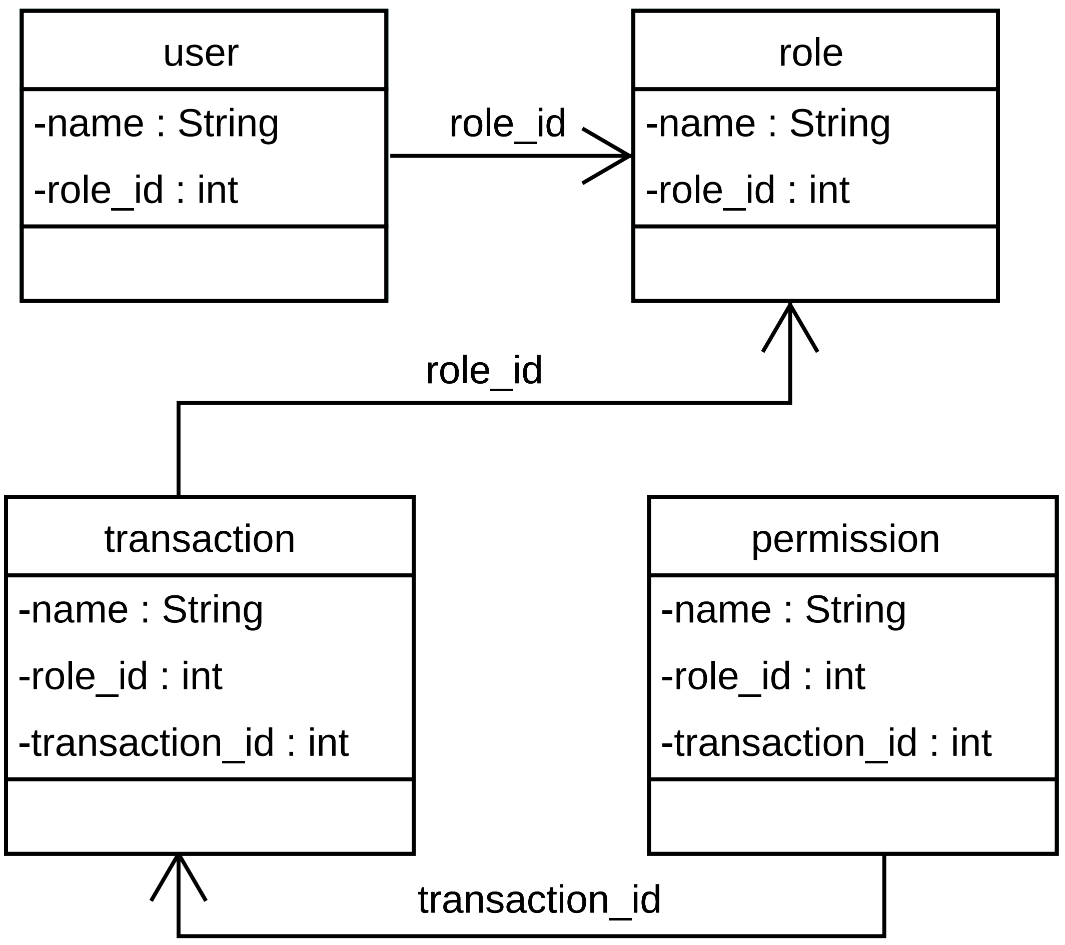

Fig. 3. Simple Role Based Access Control

constraints, for example), but in general there will be instance data. It is read by the analyser from additional XML files (for an example see Figure 4), containing a tag for every class, and within that tag another tag for each attribute. The analyser is able to generate the XML-Schema file for an UML model specified by the user, because the contents of the instance file depends on the model of the business application. With the generated Schema file the analyser is able to validate the input file.

2.6 Rules

As defined in the previous section, the business application data structure is represented by a class diagram, that is, a directed graph together with the data from the business application. These two pieces make up a rather complex graph whose structure can be seen in Figure 5 as an example. One can see that for every user in the business application data structure, a node is added. The model gives the tool the information that there is a connection between “user” and “role”, but in the graph in Figure 5 there are only edges between certain users and certain roles. It shows that there is an edge between user “john” and role “users”, because there is the attribute “role” that instantiates it. There is no edge between user “john” and

```
<rubacon>
  <user>
    <name>john</name>
    <uid>500</uid>
    <group>users</group>
  </user>
  <group>
    <group>users</group>
  </group>
...
</rubacon>
```

Fig. 4. Snippet from an instance file

role "admins", because "john" does not have "admins" in his roles. This is the graph that the analyser uses to analyse the rules.

Rules in this paper consist of the following elements:

– a name (used as a reference in the security report)
– the type of the rule, which can be either of PROHIBITION or PRECONDITION (meaning that the condition given in the sub-rule defined below should either not be fulfilled, or be fulfilled)
– a message (printed in the report if the rule fails)
– a priority level (to build a hierarchy of importance, so that less important rules can be turned off easily - typical values may include DEBUG, INFO, WARNING, ERROR, FATAL, or a numerical value)
– a sub-rule, which defines a path in the analyser's graph and a set of constraints, as defined below

A sub-rule has the following elements:

– the head, which is the starting point of the path in the analyser's graph defined by the sub-rule
– the target, which is the target of that path
– a list of constraints, which defines conditions that the path has to satisfy

Here a constraint consists of the following elements:

– element, the node that has to be checked
– condition, to be checked on that node

We consider the following example: If it has to be ensured that a certain user, say "john", does not have the role "admins" assigned, the following parameters would be set for the rule:

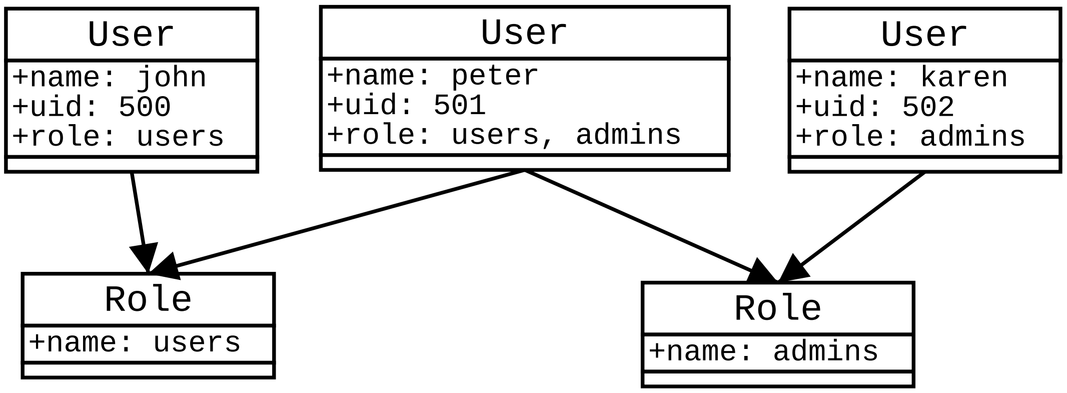

Fig. 5. The graph after model and information is inserted

name check user roles
type PROHIBITION
message check user for given roles
priority ERROR=4

In this example, we have a single sub-rule.

head user
target role
constraint head.user.name == param.user.name
constraint target.role.name == param.role.name

This rule has two parameters that the user has to provide when generating the report, indicated by the keyword *param*: the user-name "john" and the role "admins". A suitable XML document that provides these parameters for every rule is expected as input.

The evaluation of this example rule is as follows: The analyser attempts to find the head of the rule (i.e. "user: john") in the analyser's graph. Afterwards, it tries to find a path to the target (i.e. "role: admins"). If that succeeds it prints the given message in the security report, if the user wants messages with priority ERROR printed in his report. The separation between the rule itself and the two parameters ("param.user.name" and "param.role.name") is introduced to make editing more comfortable: One does not need to edit a rule for every user and every role that has to be checked.

With the help of these elements rather powerful rules can be defined. To the analyser the model is a graph representing the business application data structure. The head and the target represent nodes within that graph. For example, head could be "user" and target could be "role". With that definition there should exist a path between head and target. If it does not, the rule fails. If that path exists, the analyser will try to fill that path with valid data from the given instance-data. That means that for a valid connection from head to target, every association along that path is instantiated with a discrete entry from the business application's data. If there is no valid instantiation, the rule fails. If there is one, the constraints are checked. Every instantiated element will be examined, and if one of the conditions fails, the rule fails. Otherwise, it succeeds.

To make the rules more expressive, a rule can consist of several sub-rules, where a sub-rule does not have the additional name, type, message and level attributes. This way the analyser is powerful enough to check rules such as separation of duty, for example by using the sub-rules:

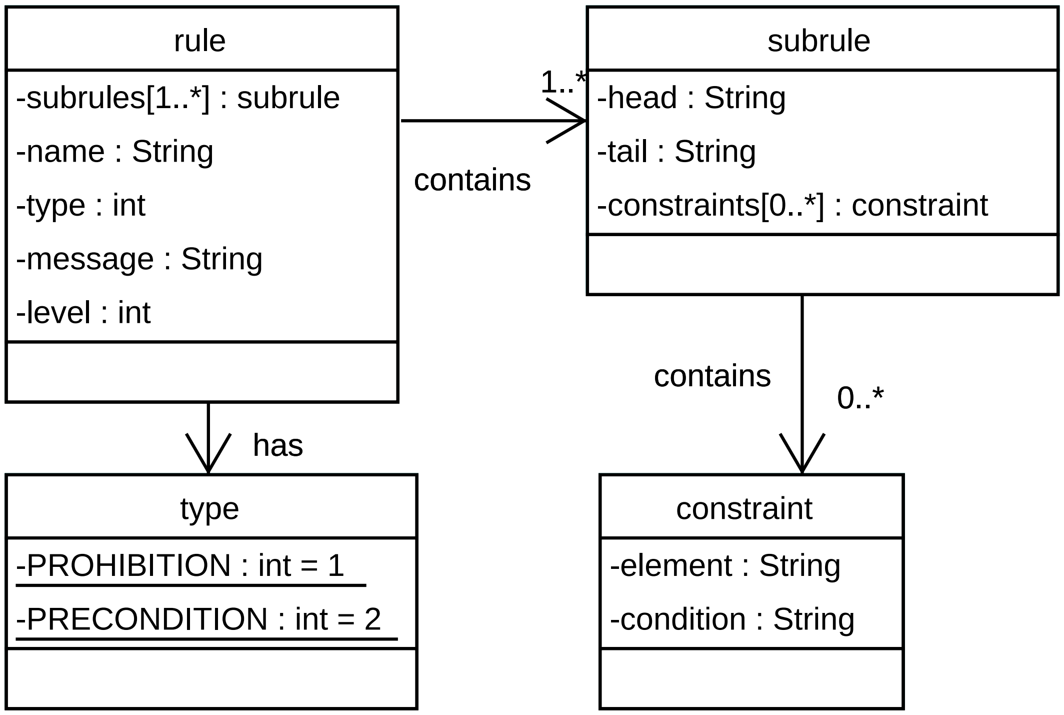

Fig. 6. Class diagram showing the structure of rules

– check for distinct role A,
– check for distinct role B and
– ensure that no user has both of them.

For a rule to succeed, each of the sub-rules has to succeed.

The additional information is needed to configure the analyser properly, and to customise the report. The name of the rule is used to output which rules failed. The type is given to distinguish between preconditions and prohibitions, meaning that either the success or failure of that rule is reported. So it is conveniently possible to define states that must be fulfilled for every configuration and to define states that may not appear within a configuration. For example, it may be vital for a system to have the password set for the super-user account. Conversely, for separation of duty, it would be forbidden for the same user to have two exclusive roles. A message is printed if a precondition fails or if a prohibition succeeds. The message attribute simply gives the text that is written to the report if a message is printed. A template system prints out the messages with any of the instance's attributes in a freely configurable manner. So it is possible to insert values from the violating instance to the message, for example as: "there is no password for user Joe". Only with such a feature the messages become readable and thus the tool easily usable by a human user.

The level-attribute gives a "level of verbosity" to a rule. So the user can have the tool evaluate some rules only. Level 1 means, that rule is relatively unimportant. An increasing number will show increasing importance for the rule. The analyser evaluates only the rules with a level higher than that given as "level of verbosity" to that report.

3 Evaluating Rules

We use Prolog for the evaluation of the rules. Prolog seems particularly suitable, because it was specifically designed for such a task. In our experience it is also sufficiently efficient for a real-life application (especially considering the possibility of over-night batch processing as in our case).

If one translates structural elements to "Atomic Prolog Terms" and the analyser-rules to "Non-Atomic Terms", one can ask the Prolog interpreter for the instances of the Prolog-rules. The advantage of using Prolog is that we can concentrate on the essential problems specific to the analyser without having to solve the hard problems of finding the instances along the paths. How the analyser-rules defined above can be translated into Prolog will be explained in the next section.

3.1 Translating rules to Prolog clauses

First of all the data structure of the business application is defined in Prolog. For this each class from the model describing the business application is converted to a predicate with an argument for each attribute. A class user (U) with two Attributes (name (n), role-id (r)) gives the following expression:

$$U(n, r).$$

To evaluate an expression like "the user's role", we need an additional predicate for each association. The "connecting" predicates have the following form: Assume there is a predicate $U(n, r)$ and a predicate $R(m, r)$. Then the connection $C(n, m, r)$ is given by the following term:

$$C(n, m, r) \rightarrow U(n, r) \wedge R(m, r).$$

That means that there is a user n in role m if there is a user n and a role m such that the role-id r is the same. These predicates can be extended to paths with any number of intermediate nodes, because Prolog evaluates all predicates to true, and the provided connecting attributes match, as in the following example:

$$A(u, v, w, x, y, z) \rightarrow B(u, v) \wedge C(v, w) \wedge D(w, x, y) \wedge E(y, z).$$

Note that the tool could be modified to eliminate the arguments not needed to determine the existence of a path. However, it is convenient to be able to include this additional information in the report.

After the structure is added, the instance will be added, too. For every class several predicates are created. In Prolog syntax that's what it looks like:

$$user(john, 500).$$

With the above rules in place, the analyser can ask for instantiations of the discrete rules (for example, if user john has exclusive roles "start transaction" and "commit transaction", separation of duty is violated for this transaction).

A short remark regarding the efficiency of the analysis: The "connecting" predicates are added only when a rule needs them. If one would insert every possible connection from every imaginable head to every target, there were up to $n(n-1)$ of these connections. But to evaluate m sub-rules one would need at most m of these connections. Thus a connection is only added when a sub-rule implies it, reducing the number of connections in general significantly.

3.2 Evaluating Separation of Duty in SAP systems

We use an example configuration from [Sch03] to explain how separation of duty in SAP systems can be evaluated by the analyser. First of all, the structure of the business application needs to be defined. For simplicity it will be assumed that the structure looks like the one presented in Figure 3. It certainly is just a very small part of the SAP security concept but as an example, it will be sufficient. There are three employees: Karen, Susan and John.

User	Role	Transaction	Permission
Karen	employee (in charge of service)	Create purchase	Is allowed to create some purchase in SAP.
Susan	employee (in charge of service, senior in rank to Karen)	Commit purchase	Is allowed to release purchase created by Karen.
John	employee purchasing agent	Place orders	Is allowed to place orders by some delivery agent.

Fig. 7. Small separation of duty example

Karen and Susan are just employees in any department, and John is a purchasing agent at the company. To have separation of duty, Karen may create a purchase and Susan may release that purchase to John. John may order the desired goods at some supplier firm. With that in place the Prolog rules would be very straight forward:

```
user(Karen, 1)
role(create-purchase, 1)
...
```

To have separation of duty in place there are two exclusive roles, which may not be assigned to the same user: "create-purchase" and "release-purchase". John just places the orders, he does not do any supervision here. The first sub-rule must have the head "user" and the target "role". The second sub-rule must have the same head and target but it needs a condition:

```
rule1.user.name == rule2.user.name
```

The type of that rule is PROHIBITION, the other attributes do not matter for this example. What does the tool do now? It has created the predicates and inserted the users and the role from the instance files. Afterwards it searches the paths for the rule. The path from user to rule is quite obvious, so the "connecting" predicate is created:

```
user_role(name, role_id, rname)
   :- user(name, role_id),
      role(rname, role_id).
```

With that predicate the rule can be evaluated to:

```
user_role_rule(name,
               role_id1, role_id2)
    :- user_role(name1, role_id1, X),
       user_role(name2, role_id2, Y),
       name1 = name2.
```

Now Prolog can be asked for

```
user_role_rule(X,
                 ‘create-purchase’,
                 ‘release-purchase’).
```

and Prolog calculates the correct answer. In the example from Figure 7 there is no solution to the predicate, because there is only Karen for role "create-purchase" and Susan for role "release-purchase", and user Karen is not equal to user Susan.

Although this examples is very simple, it serves as a demonstration of how the analyser can be used. In a real application, the path from user to role might contain several nodes or one might not know the roles that have to be exclusive, just the permissions, so one could exclude permissions contained in roles with several hundreds of entries each. In cases were a role contains several hundreds of permissions, it is not obvious whether separation of duty is in place.

Note that we do not currently aim to treat object-based permissions, but remain at the class level. While it should be possible to extend our approach in that direction, it is beyond the scope of the current investigation. In particular, this applies to a special kind of separation-of-duty specific to SAP systems: The system can be configured to require more than one user with a certain role to start a transaction. Since the checks needed to enforce this requirement are performed within the SAP system, it would not make sense to repeat them as well at the analysis level. But one should note, that this "internal" separation-of-duty differs form what is presented in our examples. It is often useful to have separation-of-duty througout different departments, so that the internal one is not sufficient (i.e. if there is some kind of revision after the transaction was performed).

3.3 SAP Transactions

Another example for a use of the analyser to improve security is, when the transactions are also part of the data structure. Because of the design of the SAP system (which may seem surprising from a security point of view), there are no security checks performed when a transaction calls another one. By this transitivity, it is very difficult in large systems to see who can execute a transaction. The permission to execute a transaction includes the permission to execute every transaction called by the first one and there does not seem to be a possibility to disable this feature. Thus creating a transaction in SAP is a permission that gives access to everything. One should notice that an employee who is allowed to create a transaction and execute it, can execute any transaction by calling it from his self-created one.

If access needs to be restricted to some transactions, it is therefore not sufficient to ensure that the permission is given only in the roles associated with that transaction, and that only the users allowed to execute that transaction are assigned those roles. It has to be ensured furthermore that there is no transaction calling the restricted one, because SAP would not perform security checks there and one would not prevent execution of the restricted transaction.

To do so, one may model the transactions with its sub-transactions as part of the analyser's model. Then the tool creates rules to check whether permissions grant any user additional rights that are not part of his role. It is usually not advisable to report every

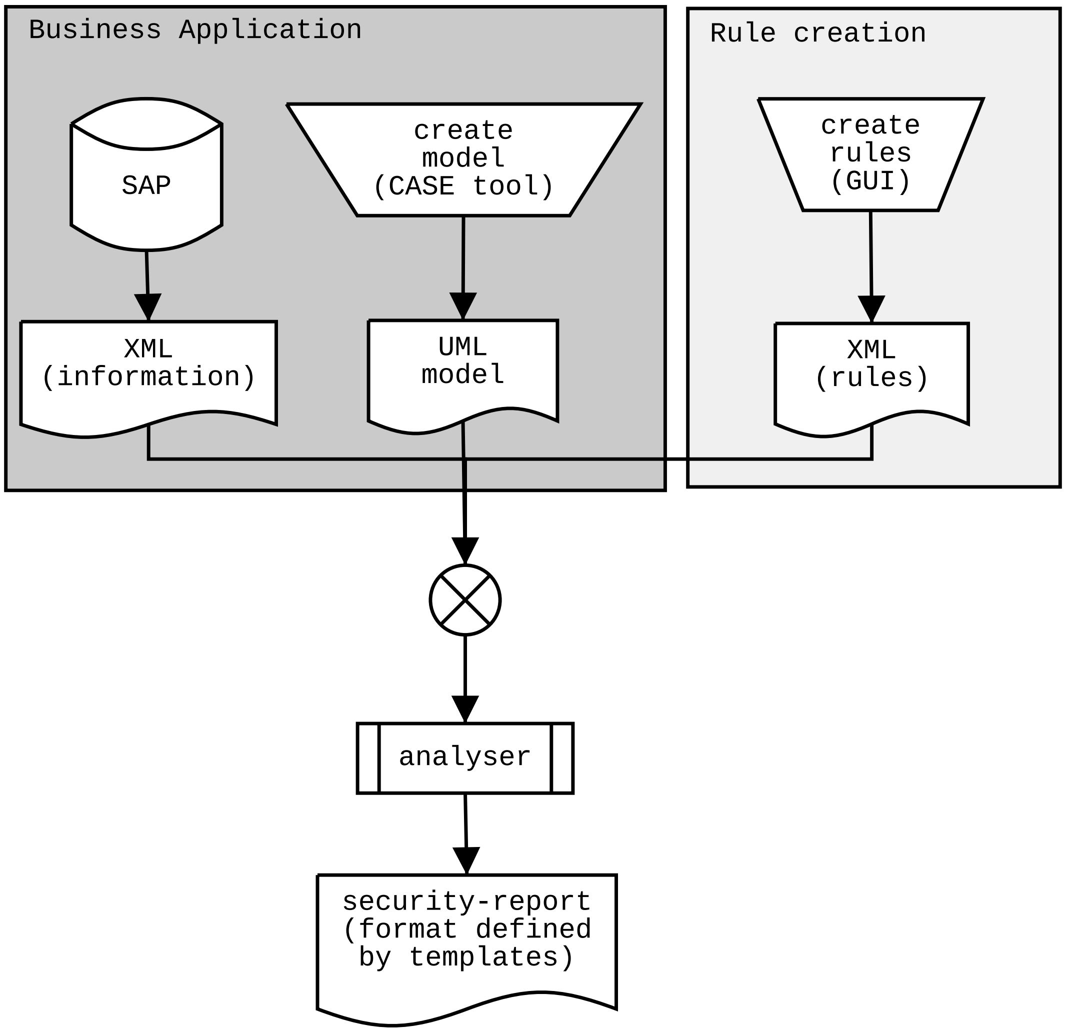

Fig. 8. Sample Configuration for Using the Analyser

transaction that can be executed without explicit permission. Because of the error-prone design, there will be a lot of transactions that are meant to be called implicitly. But the possibility to check for some very "dangerous" transactions (in particular the ones for changing permissions and roles) is a great enhancement of security. This would be improved even more if there was a way to automatically create the data that represents the dependencies of the transactions. We aim to inquire the possibilities to do so.

3.4 Use-Case to Check SAP Permissions

Figure 8 presents a sample "use-case" for checking the permissions on a running SAP system. The SAP database is used to generate the information necessary for the analyser. An employee creates an UML model describing the SAP system. We use the CASE tool Poseidon for UML to do so. These two documents describe the business application. With these documents in place one can create the rules. For creating the rules there is a graphical user interface but the XML files necessary can be edited manually, too.

When all the documents are prepared, the analyser can check the rules automatically. After the analyser has finished the checks, the user can read the security-report and start reconfiguring the business application in order to fulfil all the conditions contained in his rule-set.

The security report is formatted as defined by the templates that are part of the analyser. The analyser writes a freely configurable HTML file for review with a web browser.

3.5 Further applications

The analyser can not only be used to check SAP systems, it can be used to check most configurations of large scale applications. The modular architecture makes it easy to adapt to a new application. One needs to define the application's structure in UML, then the instance data must be converted to proper XML files, corresponding to the XML Schema provided by the tool's schema generator. Afterwards, the rules have to be defined. There the graphical user interface can be used, or the XML files can be written manually or generated by any tool fitting the needs of the application. Then the report can be generated by the analyser.

With that open architecture, we hope to establish a tool for a wide range of rule-checking tasks of configuration files. Our main focus of application is security, but there are other fields where one could use the analyser.

4 Related Work

One approach to analysing security configurations is called "Configuration Review Test" [Pol92]. As far as we know there is no implementation of these tests that uses rules for this purpose. Existing tools for this approach check some conditions of specific applications, mostly operating systems. These tools are designed to check for certain security weaknesses, common to a number of systems. Compared to this specific tools, the open architecture presented in the current paper is new for configuration review tests. We consider it a useful new idea for tool-supported security checks. The analyser presented here could also replace some of the more specific tools, by adding some applications that collect all the information necessary to check, for example, operating system's configurations.

Penetration tests are commonly used to assess the security of a system [Wei95]. In our view, they are complementary to our approach: On the one hand, penetration tests would profit from the information gathered by the analyser's report. On the other hand, the analyser presented here does not warn about weaknesses in the software itself (such as programming errors or buffer overflows), but it reports configuration errors. To have a penetration test reveal the errors, the analyser is designed to check for, one would have to try out every possible transaction. This is usually impractical because there are too many of them. Also, when performed on a live system, the penetration test would be rather invasive.

There are several recent approaches using UML for security analysis, including [Jür03,LBD02]. More generally, there has been a lot of work on formulating security requirements in object-oriented data models (see for example [JKS95] and the references there). Other approaches using logic programming for access control analysis include [BdVS02]. [RS01] uses SQL to administer permissions for distributed data. Compared to that approach, our tool can not only be applied to data bases, but more generally to security configurations. [GAR03] uses a model-checker to analyse Linux configurations.

5 Conclusion

The analyser introduced in this paper is capable of reading the business applications configuration as an UML model and a XML file, therefor it can be easily configured for a wide variety of business applications. The rules used for checking are rather flexible and powerful. While the tool uses a template system for it's report the layout of that report can be freely adopted to any form required (e.g. HTML, pdf with the help of latex, . . .).

Misconfiguration of security mechanisms is a major source of attacks in practise. The current work aims to address this issue by providing automated tool-support for checking SAP security permissions. The tool allows one to formulate rules (such as separation-of-duty) that the permissions are supposed to satisfy. It enables one to check automatically that the permissions actually implement the rules even in situations where this is difficult, laborious, and error-prone to perform by hand, because of dynamic changes and the size of the data volumes involved.

Because of its modular architecture and its standardised XML interfaces, the tool can be easily adapted to check security constraints in other kinds of application software (such as firewall or other access control configurations). By making use of standardised mechanisms (such as UML) for specifying the rules, it should be easily learnt to use.

One of the advantages of the current work in comparison to other possible approaches to the problem is the possibility to link the analysis of the SAP permissions with an analysis of a business process model given as a UML activity diagram, which is work in progress [Sch03].

A prototype of the analyser can be found at **http://www4.in.tum.de/∼hoehn** .[1]

References

[AJP95] M. Abrams, S. Jajodia, and H. Podell, editors. *Information security: an integrated collection of essays.* IEEE Computer Society Press, 1995.

[BdVS02] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security*, 5(1):1–35, February 2002.

[GAR03] J.D. Guttman, A.L.Herzog, and J.D. Ramsdell. Information flow in operating systems: Eager formal methods. In *Workshop on Issues in the Theory of Security (WITS'03).* IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[JKS95] S. Jajodia, B. Kogan, and R. Sandhu. A multilevel-secure object-oriented data model. In Abrams et al. [AJP95].

[Jür03] J. Jürjens. *Secure Systems Development with UML.* Springer-Verlag, Berlin, 2003. In preparation.

[LBD02] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 – The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, Dresden, Sept. 30 – Oct. 4 2002. Springer-Verlag, Berlin.

[Obj02] Object Management Group. Meta-object facility, version 1.4. In *OMG Specifications.* OMG, April 2002.

[Pol92] W. Timothy Polk. Automated tools for testing computer systems vulnerability. In *NIST Special Publications.* National Institute of Standards and Technology, December 1992.

[Pow02] Richard Power. 2002 CSI/FBI computer crime and security survey. Technical report, Computer Security Institute, Spring 2002.

[RS01] A. Rosenthal and E. Sciore. Administering permissions for distributed data: Factoring and automated inference. In *IFIP11.3 Conf. on Data and Application Security*, 2001.

[1] The homepage is currently under construction. For internal review use we already provide access: username: rubacon, password: rubacon.

[Sch03] Marillyn Aidong Schwaiger. Tool-supported analysis of business processes and SAP permissions, 2003. Study project, TU Munich. In preparation.

[Wei95] C. Weissman. Penetration testing. In Abrams et al. [AJP95], chapter 11, pages 269–296.