A Decade of Model-Driven Security

David Basin
ETH Zurich
Switzerland
basin@inf.ethz.ch

Manuel Clavel
IMDEA Software Institute
Universidad Complutense
Madrid, Spain
manuel.clavel@imdea.org

Marina Egea
IMDEA Software Institute
Madrid, Spain
marina.egea@imdea.org

ABSTRACT

In model-driven development, system designs are specified using graphical modeling languages like UML and system artifacts such as code and configuration data are automatically generated from the models. Model-driven security is a specialization of this paradigm, where system designs are modeled together with their security requirements and security infrastructures are directly generated from the models.

Over the past decade, we have explored different facets of model-driven security. This research includes different modeling languages, code generators, model analysis tools, and even model transformations. For example, in multi-tier systems, we used model transformations to transform a security policy, formulated for a system's data model, to a security policy governing the behavior of the system's graphical user interface. In this paper, we survey progress made, tool support, and case studies, which attest to the flexibility and power of such a multi-faceted approach to building secure systems.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: General

General Terms

Security, Languages, Design, Verification

Keywords

Model-driven security, model-driven development, model analysis, model transformation, code generation

1. INTRODUCTION

Model building is at the heart of system design. This is true in many engineering disciplines and is increasingly the case in software engineering. But model building is not an end in itself and certainly does not come for free: it takes time and knowledge to build good models and effort to keep them synchronized with end products. For this effort to be worthwhile, there must be added value.

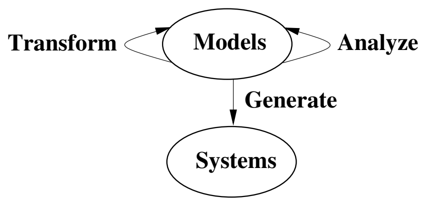


Figure 1: Use of Models in Model-Driven Security

In this paper, we examine some of the advantages over more traditional approaches to model building in the domain of security-critical systems. In particular, we survey our work [32, 9, 5, 6, 15, 7, 18] over the past decade on *model-driven security.* We show that models can be used for the following four activities in the development of secure systems:

A1. Precisely documenting security requirements together with design requirements.

A2. Analyzing security requirements.

A3. Model-based transformation, such as migrating security policies on application data to policies for other system layers or artifacts.

A4. Generating code, including complete, configured security infrastructures.

Figure 1 depicts these activities and their interrelationships. Designers specify security-design models that combine security and design requirements (A1). As our modeling languages have a well-defined semantics, we can formally analyze these designs (A2). When designing secure systems, security may be relevant at different system layers or views. Using model transformations, we can migrate a security policy from one model to other models (A3). Finally, we can use tools to automatically generate code and other system artifacts directly from the models (A4).

In the subsequent sections we explore these activities in more detail. In doing so, we highlight the central and multi-faceted role that models can play in developing secure systems. We also explain the development of our ideas, advances in tool support, and applications.

Organization. In Section 2 we introduce security-design models and give examples. In Sections 3 and 4 we present different ways to analyze and transform such models. In

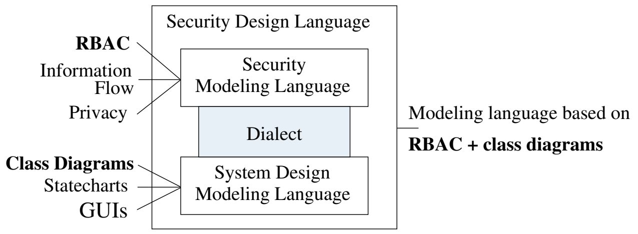


Figure 2: Model-languages and their Combination

Section 5 we describe SSG, a tool that supports the design, analysis, and transformation of security-design models for developing security-aware GUIs for data-centric applications. In Section 6 we report on our experience applying model-driven security in practice. Finally, we discuss related work in Section 7 and draw conclusions in Section 8.

2. MODELING

2.1 Languages

Model-driven security is a specialization of *model-driven development*, also called *model-driven architecture* [35], to the domain of security. The crucial part of this specialization concerns the modeling language. Instead of adopting a one-language-fits-all approach, we proposed a general schema for integrating security requirements into system design models. The main idea is to define security modeling languages that are general in that they leave open the nature of the protected resources, i.e., whether these resources are data, business objects, processes, controller states, etc. Figure 2 provides examples of different security notions which could be specified using a security modeling language (top-left) that one might integrate with different design modeling languages (bottom-left), resulting in a security-design modeling language (right side). For example, one might combine a modeling language for Role Base Access Control (RBAC) with Class Diagrams, as indicated in bold in the figure. This combination is made by defining a dialect (or "glue"), which identifies elements of the design language as the protected resources of the security language. In this way, we can flexibly define languages for formulating different kinds of system designs along with their security requirements.

In previous work, we defined a security modeling language called *SecureUML* for modeling authorization policies based on RBAC extended with constraints [9]. We initially combined SecureUML with a design modeling language based on class diagrams, called *ComponentUML*, and with a language based on state diagrams, called *ControllerUML*. We later [7] combined SecureUML with a language for modeling graphical user interfaces for data-centric applications, called *ActionGUI*.

2.2 Example

We introduce an example, which we use in this paper to illustrate the use of the modeling languages mentioned above, namely, ComponentUML, SecureUML+ComponentUML, ActionGUI, and SecureUML+ActionGUI. In subsequent sections we use this example to illustrate model-based analysis and transformation techniques.

ComponentUML is a simple language for modeling component-based systems. Essentially, it provides a subset of UML class models: *entities* can be related by *associations* and may have *attributes* and *methods*.

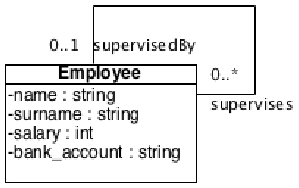


Figure 3: ComponentUML model for Employee.

In Figure 3 we use ComponentUML to model the data associated with a company's employees. In our example, an employee has a name, a surname, a salary, and a bank account. Also, an employee may possibly have a supervisor and may in turn supervise other employees. In the terminology of ComponentUML, `Employee` is an *entity*; `name`, `surname`, `salary`, and `bank account` are *attributes*, and `supervisedBy` and `supervises` are *association-ends*.

We can refine this model by adding constraints (also called *invariants*) to it. For example, we can specify that:

1. There is exactly one employee who has no supervisor.
2. Nobody is his (or her) own supervisor.

We use the Object Constraint Language (OCL) [36] to add constraints to ComponentUML models. For example, the above constraints can be formalized in OCL as follows:

```
(1) Employee.allInstances()
      ->one(e|e.supervisedBy->isEmpty())
(2) Employee.allInstances()
      ->forAll(e|e.supervisedBy->excludes(e))
```

SecureUML+ComponentUML is the combination of SecureUML with ComponentUML. As already mentioned, SecureUML extends RBAC with authorization constraints, which enable the specification of policies that depend on the system state. SecureUML leaves open what the protected resources are and which actions these resources offer to clients; both of these depend on the primitives for constructing models in the associated system-design modeling language. In the case of SecureUML+ComponentUML, the protected resources are the entities, as well as their attributes, methods, and association-ends. The actions that are offered to clients are to create or delete entities, update or read the entity's properties, and execute the entity's methods.

In Figure 4 we use SecureUML+ComponentUML to model the company's authorization policy for accessing the data associated with its employees, according to the employee data model in Figure 3. In this example, permissions are assigned to two non-disjoint sets of users: workers (any employee) and supervisors (any employee who supervises other employees). In the terminology of SecureUML, `Worker` and `Supervisor` are *roles*. Permissions in SecureUML are granted upon satisfaction of specific constraints, written in a simple extension of OCL.[1] Namely:

[1]The variables `self` and `caller` are interpreted as follows: `self` refers to the (root) resource being accessed and the variable `caller` refers to the user accessing the resource. In SecureUML+ComponentUML, the type of the variable `caller` must be an entity in the given model. In this ex-

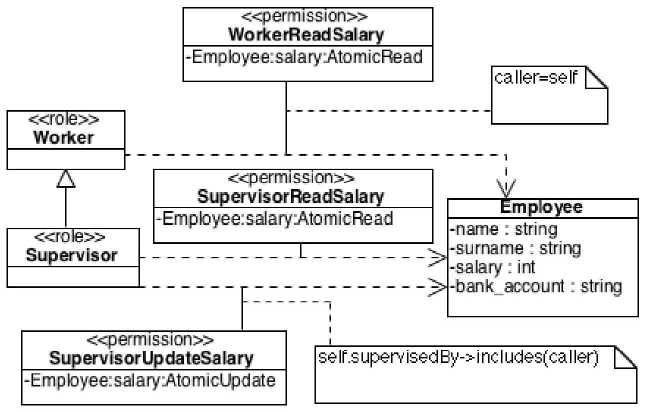


Figure 4: SecureUML+ComponentUML model for Employee.

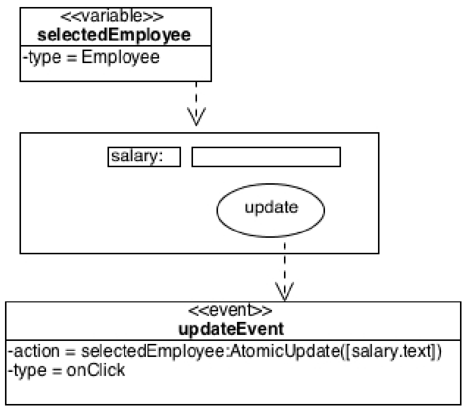


Figure 5: ActionGUI model for UpdateSalary window.

1. A worker is granted the permission to read an employee's salary (`Employee:salary:AtomicRead`), provided that it is its own salary (`caller=self`).

2. A supervisor is granted the permission to update an employee's salary (`Employee:salary:AtomicUpdate`), provided that he supervises this employee (`self.supervisedBy->includes(caller)`).

3. A supervisor is granted unrestricted permission to read any employee's salary (`Employee:salary:AtomicRead`), since no constraint is associated to this permission.

Finally, in our model, the role `Worker` generalizes the role `Supervisor`. This means that supervisors inherit all the permissions granted to workers, along with their associated authorization constraints.

Note that when modeling this authorization policy, our modeler probably has in mind the invariant (2) as a constraint on the data model: nobody can be his own supervisor. This prevents a (self-)supervisor from changing his own salary. We will return to this point in Section 3.

ActionGUI is a language for modeling graphical user interfaces (GUIs) for data-centric applications. In a nutshell, a GUI consists of *widgets*, which may be containers (e.g., windows, combo-boxes, or tables) or basic widgets (e.g., buttons, labels, or entries). A widget may have a set of associated *events* (e.g., entering or leaving a widget, creating a widget, or clicking on a widget.) Then, an event may trigger a set of (possibly conditional) *actions* on the widgets themselves (e.g., opening and closing a widget) or on the application data (e.g., reading or updating an attribute). Also, a widget may have associated *variables* which hold information to be used by the actions triggered by the events associated to the widget.

In Figure 5 we use ActionGUI to model a window for updating a (previously selected) employee's salary. The window has an entry labelled `salary`, and a button labelled `update`. In addition, it has a variable `selectedEmployee`, that holds a reference to an instance of the entity `Employee`; in this example, this is the employee whose salary will be updated. Within the window model, the modeler can refer to the string input by the user in the entry `salary` using the expression `salary.text` enclosed in square brackets. Notice that the button `update` supports an event `updateEvent`. In ActionGUI, each event has a type and, in our example, `updateEvent` has the type `onClick`. In this example, when the user clicks the `update` button, the salary of the `selectedEmployee` is updated (`selectedEmployee:salary:AtomicUpdate`) using as the parameter the string typed by the user in the entry `salary` (`[salary.text]`).

ample, the type of the variable `caller` is `Employee`, which means that the users accessing the resources will be employees, i.e., instances of the entity `Employee`.

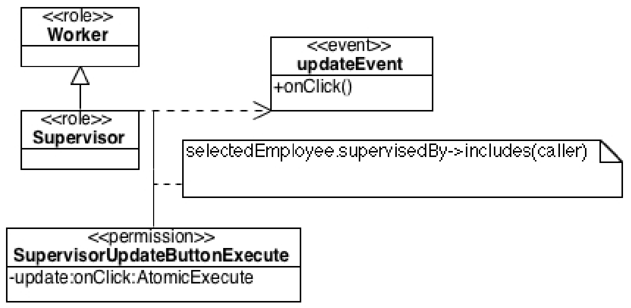
SecureUML+ActionGUI is the combination of SecureUML with ActionGUI. It provides a language for modeling security-aware GUIs. In the case of SecureUML+ActionGUI, the protected resources are the widgets, and the actions that are offered to clients are to execute the widgets's events.

In Figure 6 we use SecureUML+ActionGUI to model the authorization policy for executing the event `updateEvent` supported by the button `update` in the window modeled in Figure 5. In SecureUML+ActionGUI, authorization constraints are written using an extension of OCL where, as in the case of SecureUML+ComponentUML, the variable `caller` refers to the user who is accessing the resource (in this case, executing the event). In this model, only supervisors are granted permission to click on the `update` button (`update:onClick:AtomicExecute`) and, furthermore, they may only do this when they supervise the `selectedEmployee` (`selectedEmployee.supervisedBy->includes(caller)`).

The SecureUML+ActionGUI model in Figure 6 raises a number of "consistency" questions with respect to the SecureUML+ComponentUML model in Figure 4. For example, could a user who is not authorized to update the selected employee's salary still be permitted to click on the `update` button? Alternatively, could a user with this authorization fail to have permission to click on the `update` button? We will return to these questions in Section 3. On a more practical level, a question that also arises is whether it would be possible to automatically generate the model in Figure 6 from the models in Figures 4 and 5. We will explain how this can be done in Section 4.

3. ANALYSIS

Security-design models are formal objects and therefore one can reason about their properties. In our previous work, we have proposed formal techniques to answer questions



<<role>>
Worker
<<event>>
updateEvent
+onClick()
<<role>>
Supervisor
selectedEmployee.supervisedBy->includes(caller)
<<permission>>
SupervisorUpdateButtonExecute
-update:onClick:AtomicExecute

Figure 6: SecureUML+ActionGUI model for UpdateSalary window.

about the meaning of security-design models at two different levels:

1. **The models themselves.** Here the questions are about the elements that are contained in a model and their relationships. For example, given a role, what are the actions that a user in this role can perform? Are there overlapping permissions for different roles? Are there two roles that have permission to perform the same set of actions?

2. **The models' instances.** Here the questions are about the scenarios that are consistent (or conformant) with the model, i.e., about all the valid instances of the model. For example, is there a scenario consistent with the model in which someone satisfying a property P_1 is allowed to perform an action on resources satisfying some other property P_2?

To answer questions about the models themselves in a semantically precise and meaningful way, we proposed in [6] a metamodel-based methodology consisting of: (i) formalizing these questions as queries in OCL, in the context of the metamodel combining SecureUML with the design language; and (ii) evaluating these OCL queries on the instance of the metamodel which corresponds to the model being analyzed. With respect to (i), [6] provides examples of non-trivial OCL queries about SecureUML+ComponentUML models. With respect to (ii), as reported in [13], current OCL evaluators can automatically answer complex queries about large collections (including up to a million objects) in less than 5 seconds. For example, in [6], an operation `allAtomics()` is defined in OCL that, given a role, returns the collection of atomic actions that a user in this role can perform upon the satisfaction of any associated constraints. Thus, by evaluating the expression `Supervisor.allAtomics()`, we can obtain all the atomic actions that supervisors are allowed to perform (again, upon satisfaction of the associated constraints) according to the model in Figure 4, namely, `{Employee:salary:AtomicRead, Employee:salary:AtomicUpdate}`. Recall that the expression `Supervisor.allAtomics()` is to be evaluated on the instance of the metamodel of SecureUML+ComponentUML corresponding to the model in Figure 4.

To answer questions about the model instances that are consistent with a given security-design model, we need to go beyond simple query evaluation and resort to theorem-proving techniques. Consider, for example, the access control policy model shown in Figure 4. Is the scenario in which someone is allowed to change his own salary consistent with this model? First, by querying the model as explained above, we learn that only supervisors (i.e., users in the role `Supervisor`) can change employees' salaries (i.e., execute the action `Employee:salary:AtomicUpdate`). Second, by querying this model again, we see that supervisors can change employees' salaries only when these employees are under their supervision, as stated by the authorization constraint

```
self.supervisedBy ->includes(caller)
```

which is associated to the permission `SupervisorUpdate-Salary`. Therefore, we can rephrase our initial query as: is there a scenario in which the following OCL expression evaluates to true?

```
(3) Employee.allInstances()->exists(self, caller|
     self.supervisedBy->includes(caller)
     and self=caller)
```

Recall that we intend to forbid self-supervision by imposing the invariant (2) on the employee data model. To answer our initial question (can someone change his own salary?) it is therefore sufficient to prove that for any scenario consistent with (2) then (3) must be false.

We currently answer such questions using a mapping from OCL to first-order logic that we introduced in [14]. Using this mapping, we can reformulate these questions as satisfiability problems in first-order logic and use theorem-proving tools (including SMT solvers, as reported in [14]) to automatically answer them. In Appendix A we give the satisfiability problem corresponding to this example, formulated in the syntax of the SMT solver Yices [19]. The answer automatically provided by Yices is `unsat`. Hence (2) and (3) cannot both be simultaneously true in any scenario of the model shown in Figure 3.

Another interesting question that can be answered using our mapping from OCL to first-order logic is the following: if an employee has no supervisor, is it possible at all to update his salary? This question can be rephrased as: is there a scenario in which the following expression evaluates to true?

```
(4) Employee.allInstances()->exists(self, caller|
     and self.supervisedBy->includes(caller)
     self.supervisedBy->isEmpty())
```

As before, the answer automatically provided by Yices is `unsat`. Thus (4) cannot be true in any scenario of the model shown in Figure 4.

4. TRANSFORMATION

In model-driven development, transformation is the way of using models to produce other development artifacts. The following types of transformations have been widely used in the model-driven development community.

Generation of Code and Execution Artifacts: Models may be mapped to code or other artifacts that affect the system's runtime behavior. When generating code, the transformation function amounts to a kind of translator or compiler. Examples of other artifacts generated are deployment and configuration data (e.g., for access control on an application server, database, firewall, operating system, etc.), which also affects the system's behavior.

Generation of Models: Models may be mapped to other models. In particular, when multiple models serve as input, one speaks of a many-models-to-model transformation. We will give an example of this shortly.

Generation of Test Cases: Test cases can be generated from models, e.g., to test an access control policy.

Below we elaborate on the first two possibilities, which we have pursued in model-driven security.

4.1 Code Generation

Originally [32, 8], we mainly used security-design models for generating code along with deployment and configuration data. We built translators that map models into access control infrastructures for distributed object-based systems. In particular, we built generators for systems conforming to the Enterprise JavaBeans (EJB) standard and Microsoft Enterprise Services for .NET. From models specifying secure components and secure controllers we generated access control infrastructures for multi-tier web applications. These ideas were integrated into several tools, both academic and commercial, e.g., the ArcStyler tool [28] of Interactive Objects GmbH.

We briefly describe the ideas behind code generation for security-design models in SecureUML+ComponentUML for an EJB platform. Given a model like that of Figure 4, our code generator produces Java code and access control configuration data stored in an XML configuration file (called a *deployment descriptor*). A class like `Employee`, which represents a persistent entity, is transformed to an EJB component of type *entity bean* with all necessary interfaces and an implementation class. Namely, each method in the class is transformed to a method declaration in the component interface of the respective entity bean and a method stub in the corresponding bean implementation class. Also, for each attribute, "getter" and "setter" access methods are generated for reading and writing the attribute value; association-ends are handled analogously.

The above describes routine code generation for the design part of a security-design model. For the security part, roles and permissions are mapped into an EJB security infrastructure based on RBAC. In particular, our translator maps the roles and permissions into XML formalizing which roles have access to which actions on which resources. Authorization constraints are translated to Java assertions that check these constraints at run time. For example, in Figure 4, we would generate the following RBAC configuration data for the permission `WorkerReadSalary`: anyone in the roles `Worker` or `Supervisor` can read the attribute `salary` in the `Employee` class, i.e., can execute the getter-method that reads the value of the attribute `salary`. Moreover, given the authorization constraint, we would generate an assertion, placed at the start of this getter method, that checks that the authenticated caller is the employee whose salary is being read, when the caller belongs to the role `Worker`.

4.2 Model Transformation

It is also possible to transform models into other models. Typically such transformations add details, specialize constructs, or change representations. An example of this is the specialization of platform-independent models to platform-specific models. In our work, we have explored a particular application of model transformation: how to consistently apply a security policy to multiple system layers. We see this as an important part of building effective security infrastructures, as we shall now explain.

Security is often built redundantly into systems. For example, in a web-application, access control may be enforced at all tiers: at the web server, in the back-end databases, and even in the GUI. There are good reasons for this. Redundant security controls is an example of defense in depth and is also necessary to prevent data access in unanticipated ways, for example, directly from the database thereby circumventing the web application server. Note that access control on the client is also important, but more from the usability rather than the security perspective. Namely, although client-side access control may be easy to circumvent, it enhances usability by presenting honest users an appropriate view of their options: unauthorized options can be suppressed and users can be prevented from entering states where they are unauthorized to perform any action, e.g., where their actions will result in security exceptions thrown by the application server or database.

This raises the following question: must one specify security policies separately for each of these tiers? The answer is "no" for many applications. Security can often be understood in terms of the criticality of data and an access control policy on data can be specified at the level of component (class) models, as discussed in Section 2. Afterwards, an access control policy modeled at the level of components may be lifted to other tiers. When the tiers are also modeled, this lifting can be accomplished using model transformation techniques and in a precise and meaningful way.

In our previous work [37, 7, 18] we have taken such a transformation-based approach to systematically lift security policies specified on data to policies governing graphical user interfaces. Under our proposal, the process of modeling a security-aware GUI has the following four parts.

1. Software engineers specify the application-data model.

2. Security engineers specify the security-design model.

3. GUI designers specify the application GUI model.

4. A many-models-to-model transformation automatically generates a security-aware GUI model from the security model and the GUI model.

We have implemented this transformation-based approach for generating security-aware GUIs using the Operational Query/View/Transformation (QVT) engine [20]. The many-models-to-model transformation at the core of this approach is ultimately defined in terms of *data actions*, since data actions are both controlled by the security policy and triggered by the events supported by the GUI.

Specifically, to generate a SecureUML+ActionGUI model, our model transformation proceeds in two steps.

Step 1: The model elements of the target model are created using the elements from the two source models. In particular, each role in the (source) SecureUML+ComponentUML model is copied, along with its generalization associations, to the (target) SecureUML+ActionGUI model. Afterwards, each widget in the (source) ActionGUI model is copied, along with its associated events and actions, to the (target) SecureUML+ActionGUI model.

Step 2: The permission assignments in the target model are created. Namely, for each role and each event in the (target) SecureUML+ActionGUI model, when the users in the role are allowed to perform all the data actions triggered by the event according to the (source) SecureUML+ComponentUML model, then a permission is created in the target model. This permission grants access to the role to execute the event, upon the satisfaction of the corresponding authorization constraints.[2]

Consider, for example, the security-design model and the GUI model shown in Figures 4 and 5. Using our many-models-to-model transformation, we automatically generate the security-aware GUI model shown in Figure 6. This model includes both the roles `Worker` and `Supervisor`, as well as the permission for the latter to execute the event of type `onClick` on the `update` button. Recall that this will trigger the action of updating the salary of the `selectedEmployee`. So, as expected, this permission is constrained by the following authorization: `selectedEmployee.supervisedBy->includes(caller)`. Hence, to be authorized to click on the `update` button, the supervisor must be among the supervisors of the `selectedEmployee`.

In general, model transformations support problem decomposition during development where design aspects can be separated into different models which are later composed. As a methodology for designing security-aware GUIs, this approach supports the consistent propagation of a security policy from component models to GUI models and, via code generation, to GUI implementations. This decomposition also means that security engineers and GUI designers can independently model what they know best and maintain their models independently.

5. TOOL SUPPORT

As part of our work, we have developed a software-development environment, called SSG [18] for building security-aware graphical user interfaces for data-centric applications. SSG provides tool support for the four activities depicted in Figure 1: modeling, analysis, transformation, and generation.

SSG consists of a collection of plugins that have been developed for Eclipse. First, SSG contains three different editors for graphical modeling, which has been developed using the Eclipse Graphical Modeling Framework. These are (i) an editor for modeling application data using ComponentUML; (ii) an editor for modeling the application's access control policy on data using SecureUML+ComponentUML; and (iii) an editor for modeling the application's graphical user interface using ActionGUI.

Second, SSG contains a plugin that allows modelers to use OCL to query their SecureUML+ComponentUML models, which corresponds to the first kind of model analysis described in Section 3. This plugin includes an OCL parser and an OCL evaluator [13], the latter serves to automatically answer queries about the elements (roles, permissions, authorization constraints, actions, and resources) contained in models and their relationships. Moreover, a plugin is currently under construction to support the second kind of model analysis described in Section 3, namely, queries about the instances of SecureUML+ComponentUML models, using the mapping from OCL to first-order logic proposed in [14].

Third, SSG contains a QVT transformation that automatically performs the many-models-to-model transformation described in Section 4.2. It automatically transforms an ActionGUI model and a SecureUML+ComponentUML model (both sharing the same ComponentUML data model) into a SecureUML+ActionGUI model. The resulting model has the same behavioral properties as the one modeled by the given ActionGUI model, except that it is now security-aware with respect to the access control policy modeled by the SecureUML+ComponentUML model.

Finally, SSG includes a code generator, based on Java Emitter Templates (JET) [21], that automatically generates a full web application from a SecureUML+ActionGUI model. This application consists of a collection of PHP-web pages whose design and behavior implement those specified by the given model. In particular, windows are implemented as web pages. Thus, opening a window is implemented as loading the corresponding page and closing a window is implemented as loading the previously visited page. More interestingly, data actions (like creating or deleting entities and updating or reading their attributes) are implemented as MySQL statements on a data-base implementing the underlying ComponentUML data model. The code generator can also create this database for the user. Finally, permissions to execute events on widgets (like clicking a button or creating an entry or a text box) are implemented by conditional statements in the PHP-code responsible for interpreting those events. A key component of the SSG's code generator is a MySQL code generator [22, 17] for OCL, which translates into MySQL the OCL authorization constraints in the given SecureUML+ActionGUI model.

6. EXPERIENCE

In [15] we report on an industrial pilot project for assessing the benefits of model-driven security when applied to concrete software development projects. The project's goal was to enhance a test report configuration utility, developed in-house, with an access control policy. We used ComponentUML to model the functional requirements of the utility, and SecureUML+ComponentUML to model its access control policy. These requirements were provided to us in a five-page document listing fifty clauses in plain English. For example, users could choose from a pool of available test report configurations, which may include private, global, and default ones. Default configurations are in turn associated with individual test programs or with families of test programs. Since the permissions to create, edit, delete, or apply report configurations depended both on the user's role and on the properties (including the ownership) of the configurations, we extensively used OCL to formalize the corresponding authorization constraints.

Our experience in this project was very positive. Our security-design models helped us to understand (and discuss) the original requirements document by allowing us to independently model each clause based on its principal concern, whether functional or security-related. Their analysis

[2]Notice, however, that for these authorization constraints to be meaningful in the target model, they must also be transformed. Essentially, the variable `self`, which in the (source) SecureUML+ComponentUML refers to the data element being accessed by the user, must be replaced by the OCL expression that denotes, in the (source) ActionGUI model, the subject of the action triggered by the event.

prompted us to refine those requirements that were ambiguous, to eliminate those that were subsumed by others, and to discover those that were simply missing. Moreover, the models also provided a basis for refinement down to code. Overall, the use of security-design modeling languages provided the focal point for integrating security engineering into a model-driven software development process.

Another substantial case study was that of Lodderstedt [31], who used the model-driven security approach to construct secure web portals. As an extended example, he developed a secure version of the J2EE "Pet Store" application, which is a prototypical e-commerce application that demonstrates the use of the J2EE platform. The application features web front-ends for shopping, administration, and order processing. His application model consisted of 30 components and several front-end controllers. Lodderstedt extended this model with an access control policy formalizing the principle of least privilege, where a user is given only those access rights that are necessary to perform a job. The modeled policy comprised six roles and 60 permissions, 15 of which were restricted by authorization constraints. The corresponding infrastructure was generated automatically and consisted of roughly 5,000 lines of XML (overall application: 13,000) and 2,000 lines of Java source code (overall application: 20,000). This large expansion was due to the high level of abstraction provided by the security-design modeling languages used. Clearly, this much information cannot be managed practically at the source-code level.

7. RELATED WORK

Over the last decade, there has been substantial research on model-driven security. Here we report on related work in modeling, analysis, and transformation.

Modeling.

Numerous researchers have explored the use of UML-like languages for modeling role-based access control policies [1, 12, 11, 39, 3] and different kinds of security-design models [29, 16]. In the language UMLsec [29], for example, UML models are annotated with security requirements, such as confidentiality or secure information flow. Another prominent example is the Ponder specification language [16], which supports the rule-based formalization of authorization policies. As in the case of SecureUML [9], privileges may be organized similar to RBAC and rules can be restricted by conditions expressed in a subset of OCL. Ponder policies can be directly interpreted and enforced by a policy management platform. Model-driven security has also been employed for the development of secure XML databases [40, 24], secure databases, and data warehouses [23].

There has also been much work on integrating security requirements in process models and we mention several representative examples. In [8], we have combined SecureUML with a process design language to generate security architectures for distributed applications. SECTET [27, 4] is an extensible framework for designing and managing security-critical workflows based on web services. Finally, [43] describes a security policy and policy constraint modeling language that captures security requirements for business processes. In this work, security-annotated business processes can be translated into platform-specific target languages, such as XACML or AXIS2 security configurations.

Analysis.

There has been considerable work in analyzing the security of system designs within the Formal Methods communities, e.g., [41, 10, 30] to name a few examples. Usually traditional formal methods are used, based on model checking or theorem proving. Moreover, various groups have proposed approaches and associated tools for directly reasoning about access control policies [45, 46], including policies specified in standardized languages such as XACML [25].

With respect to the analysis of UML models, our use of OCL as a query language was inspired by [2] who used OCL to query RBAC policies; see also [38, 42]. One of the interesting challenges in our setting is that reasoning about security-design models involves different kinds of deduction problems as indicated in Section 3. This includes answering queries on models, which amounts to querying a potentially very large, but finite, scenario, and determining the existence of scenarios satisfying constraints, which calls for theorem proving or the use of constraint solvers. Model checkers or theorem provers are needed when reasoning about the combination of SecureUML with dynamic process-oriented models, such as those considered in [8].

Transformation.

Yie et al. [44] propose using model transformation-based techniques to integrate requirements, including security, in a model-driven software product line. In their setting, abstract design models of the application and of its security policy are built and refined using model transformation to obtain an implementation model with Java Platform, Enterprise Edition (JEE) security annotations. A related approach, using aspect-oriented programming, is outlined by Fox and Jürjens [26]. They propose to enrich a data model with a security policy by performing a model transformation using the bidirectional object-oriented transformation language (BOTL) [33, 34].

Finally, creating user interfaces is a common and time consuming task in application development. There have been numerous proposals and tools that aim to reduce the effort required to build effective, user-friendly graphical interfaces. Surprisingly, there has been no prior research on the systematic design of GUIs whose functionality should adhere to the security policy of the underlying application-data model.

8. PERSPECTIVE AND OUTLOOK

The ever-growing development and use of information and communication technologies is a constant source of security and reliability problems. Clearly we need better ways of developing software systems and approaching software engineering as a well-founded engineering discipline.

In model-driven development, models are the cornerstone of software and system development and can be used to abstract away irrelevant details, rigorously specify the interplay between security and functional requirements, and provide a basis for analysis and transformation. Proponents of model-driven development have in the past been guilty of making overambitious claims: positioning it as the Holy Grail of software engineering where modeling completely replaces programming in that systems are entirely generated from high-level models, each one specifying a different view of the same system. This vision is, of course, unrealizable in its entirety for simple complexity-theoretic reasons. If the modeling languages are sufficiently expressive then basic problems such as the consistency of the different models/views of a system becomes undecidable.

The original vision of model-driven security was to provide a way for software engineers to bridge the gap from security and design requirements to systems by taking a model-centric approach. This in turn necessitated bridging the gap between security modeling languages and design modeling languages, leading to the notion of security-design modeling languages, such as SecureUML+ComponentUML. Model-driven security has enormous potential not because it tackles the deep problem of synthesizing "business logic" but rather the shallow yet often extremely wide problem of generating security infrastructure. This infrastructure can be built from standard APIs and assertions and its complexity lies, essentially, in getting the deployment information right, despite the numerous details that must be considered. Security-design models provide a clear, declarative, high-level language for specifying these details. The strength of security-design models also lies in their well-defined semantics. This opens up a range of exploitation options and so far we have only scratched the surface of what is possible.

Our past work has focused primarily on access control. However, many systems have security requirements that go beyond access control, for example, obligations on how data must or must not be used once access is granted. We are currently working on handling usage control policies in the context of model-driven security. The challenge here is to define modeling languages that are expressive enough to capture these policies, support their formal analysis, and provide a basis for generating infrastructures to enforce or, at least, monitor these policies.

More generally, there are many challenging questions on the analysis side. Here, our goal is to be able to analyze the consistency of different system views. For example, suppose that access control is implemented at multiple tiers (or levels) of a system, e.g., at the middle tier implementing a controller for a web-based application and at the back-end persistence tier. If the policies for both of these tiers are formally modeled, we would like to answer questions like "will the controller ever enter a state in which the persistence tier throws a security exception?" Note that with advances in model transformations, perhaps such questions will some day not even need to be asked, as we can uniformly map a security policy across models of all tiers.

Ultimately we see model-driven security playing an important role in the construction and certification of critical systems. For example, certification under the Common Criteria requires models for the higher Evaluation Assurance Levels. Model-driven security provides many of the ingredients needed: models with a well-defined semantics, which can be rigorously analyzed and have a clear link to code. As the acceptance of model-driven development techniques spread, and as they become better integrated with well-established formal methods that support a detailed behavioral analysis, such applications should become a reality.

Acknowledgements

This work is partially supported by the EU FP7-ICT-2009.1.4 Project No. 256980, NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems. We gratefully acknowledge our previous and current collaborators in this effort: Torsten Lodderstedt, Jürgen Doser, Michael Schläpfer, Miguel A. García de Dios, Carolina Dania, Javier Valdazo, and Gonzalo Ortiz.

9. REFERENCES

[1] G.-J. Ahn and M. E. Shin. UML-based representation of role-based access control. In *Proceedings of the 9th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'00)*, pages 195–200. IEEE Computer Society, June 2000.

[2] G. J. Ahn and M. E. Shin. Role-based authorization constraints specification using object constraint language. In *Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'01)*, pages 157–162. IEEE Computer Society, 2001.

[3] M. Alam, M. Hafner, and R. Breu. Constraint based role based access control in the SECTET framework: A model-driven approach. *Journal of Computer Security*, 16(2):223–260, 2008.

[4] M. Alam, J. Seifert, and X. Zhang. A model-driven framework for trusted computing based systems. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC'07)*, pages 75–87. IEEE Computer Society, 2007.

[5] D. Basin, M. Clavel, J. Doser, and M. Egea. A metamodel-based approach for analyzing security-design models. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil, editors, *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS '07)*, volume 4735 of *LNCS*, pages 420–435. Springer-Verlag, 2007.

[6] D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. *Information and Software Technology*, 51(5):815–831, 2009.

[7] D. Basin, M. Clavel, M. Egea, and M. Schläpfer. Automatic generation of smart, security-aware GUI models. In F. Massacci, D. S. Wallach, and N. Zannone, editors, *Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS'10)*, volume 5965 of *LNCS*, pages 201–217, Pisa, Italy, 2010. Springer.

[8] D. Basin, J. Doser, and T. Lodderstedt. Model driven security for process-oriented systems. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT '03)*, pages 100–109. ACM Press, 2003.

[9] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[10] D. Basin, H. Kuruma, K. Miyazaki, K. Takaragi, and B. Wolff. Verifying a signature architecture: A comparative case study. *Formal Aspects of Computing*, 19(1):63–91, March 2007.

[11] R. Breu, G. Popp, and M. Alam. Model based development of access policies. *International Journal on Software Tools for Technology Transfer*, 5:457–470, 2007.

[12] C. Burt, B. Bryant, R. Raje, A. Olson, and M. Auguston. Model driven security: Unification of authorization models for fine-grain access control. In *Proceedings of the 7th International Enterprise Distributed Object Computing Conference (EDOC'03)*, pages 159–172. IEEE Computer Society, 2003.

[13] M. Clavel, M. Egea, and M. A. G. de Dios. Building an efficient component for OCL evaluation. *Electronic Communications of the EASST*, 15, 2008.

[14] M. Clavel, M. Egea, and M. A. G. de Dios. Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST*, 24, 2009.

[15] M. Clavel, V. Silva, C. Braga, and M. Egea. Model-driven security in practice: An industrial experience. In I. Schieferdecker and A. Hartman, editors, *Proceedings of 4th European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA '08) - Industrial Track*, volume 5095 of *LNCS*, pages 327–338, Berlin-Germany, 2008. Springer-Verlag.

[16] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In M. Sloman, J. Lobo, and E. C. Lupu, editors, *Policies for Distributed Systems and Networks (POLICY' 01)*, volume 1995 of *LNCS*, pages 18–38, Bristol-United Kingdom, 2001. Springer-Verlag.

[17] C. Dania and M. Egea. The MySQL4OCL code generator, 2010. `http://www.bm1software.com/mysql-ocl/`.

[18] M. A. G. de Dios, C. Dania, M. Schläpfer, D. Basin, M. Clavel, and M. Egea. SSG: A model-based development environment for smart, security-aware GUIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 2, pages 311–312, Cape Town-South Africa, 2010. ACM.

[19] B. Dutertre and L. Moura. Yices: An SMT solver. `http://yices.csl.sri.com/`, 2008.

[20] Eclipse Model to Model (M2M) Project. The operational QVT transformation engine. `http://www.eclipse.org/modeling/m2m/`, 2011.

[21] Eclipse Model to Text (M2T) Project. The Java emitter template (JET) framework for code generation. `http://www.eclipse.org/modeling/m2t/`, 2011.

[22] M. Egea, C. Dania, and M. Clavel. MySQL4OCL: A stored procedure-based MySQL code generator for OCL. *Electronic Communications of the EASST*, 36, 2010.

[23] E. Fernandez-Medina, J. Trujillo, and M. Piattini. Model driven multidimensional modeling of secure data warehouses. *European Journal of Information Systems*, pages 374–389, 2007.

[24] E. Fernández-Medina, J. Trujillo, R. Villarroel, and M. Piattini. Developing secure data warehouses with a UML extension. *Information Systems*, 32:826–856, September 2007.

[25] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 196–205. ACM, 2005.

[26] J. Fox and J. Jürjens. Introducing security aspects with model transformations. In *Proceedings of the 12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS'05)*, pages 543–549, Washington, DC, USA, 2005. IEEE Computer Society.

[27] C. Haley, J. Moffet, R. Laney, and B. Nuseibeh. A framework for security requirements engineering. In *Proceedings of the 2006 Software Engineering for Secure Systems Workshop (SESS'06)*, pages 35–42, New York, USA, 2006. ACM.

[28] R. Hubert. *Convergent Architecture: Building Model Driven J2EE Systems with UML*. John Wiley & Sons, 2001.

[29] J. Jürjens. UMLsec: Extending UML for secure systems development. In J. M. Jézéquel, H. Hussmann, and S. Cook, editors, *Proceedings of the 5th International Conference on the Unified Modeling Language (UML'02)*, volume 2460 of *LNCS*, pages 412–425. Springer-Verlag, 2002.

[30] G. Klein et al. sel4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[31] T. Lodderstedt. *Model Driven Security, from UML Models to Access Control Architectures*. PhD thesis, Unversity of Freiburg, Germany, 2003.

[32] 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, *Proceedings of the 5th international Conference on the Unified Modeling Language: Model Engineering, Concepts, and Tools (UML'02)*, volume 2460 of *LNCS*, pages 426–441. Springer-Verlag, 2002.

[33] F. Marschall and P. Braun. Model transformations for the MDA with BOTL. Technical report, University of Twente, 2003.

[34] F. Marschall and P. Braun. Bidirectional object oriented transformation language (BOTL). `http://sourceforge.net/projects/botl/`, 2005.

[35] Object Management Group. Model driven architecture guide v. 1.0.1. Technical report, OMG, 2003. OMG document available at `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[36] Object Management Group. *Object Constraint Language specification Version 2.2*, February 2010. OMG document available at `http://www.omg.org/spec/OCL/2.2`.

[37] M. Schläpfer, M. Egea, D. Basin, and M. Clavel. Automatic generation of security-aware GUI models. In A. Bagnato, editor, *Proceegings of the 1st European Workshop on Security in Model Driven Arquitecture (SEC-MDA'09)*, pages 42–56, Enschede, the Netherlands, 2009. CTIT Workshop Proceedings WP09-06.

[38] K. Sohr, G. J. Ahn, M. Gogolla, and L. Migge. Specification and validation of authorisation constraints using UML and OCL. In S. di Vimercati, P. Syverson, and D. Gollmann, editors, *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS '05)*, volume 3679 of *LNCS*, pages 64–79. Springer-Verlag, 2005.

[39] K. Sohr, T. Mustafa, X. Bao, and G.-J. Ahn.

Enforcing role-based access control policies in web services with UML and OCL. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)*, pages 257–266, Washington DC, USA, 2008. IEEE Computer Society.

[40] B. Vela, E. Fernandez-Medina, E. Marcos, and M. Piattini. Model driven development of secure XML databases. *ACM Sigmod Record*, 35(3):22–27, 2006.

[41] D. von Oheimb and V. Lotz. Formal security analysis with interacting state machines. In D. Gollmann, G. Karjoth, and M. Waidner, editors, *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS'02)*, volume 2502 of *Lecture Notes in Computer Science*, pages 212–228. Springer Berlin / Heidelberg, 2002.

[42] H. Wang, Y. Zhang, J. Cao, and J. Yang. Specifying role-based access constraints with object constraint language. In *Proceedings of the 6th Asia-Pacific Web Conference (APWeb '04)*, volume 3007 of *LNCS*, pages 687–696. Springer-Verlag, 2004.

[43] C. Wolter, M. Menzel, A. Schaad, P. Miseldine, and C. Meinel. Model-driven business process security requirement specification. *Journal of Systems Architecture*, 55(4):211–223, 2009.

[44] A. Yie, R. Casallas, D. Deridder, and R. V. D. Straeten. Multi-step concern refinement. In *Proceedings of the 2008 AOSD workshop on Early Aspects (EA-AOSD'08)*, pages 1–8, New York, NY, USA, 2008. ACM.

[45] N. Zhang, M. Ryan, and D. Guelev. Evaluating access control policies through model checking. *Information Security*, pages 446–460, 2005.

[46] N. Zhang, M. Ryan, and D. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1):1–61, 2008.

APPENDIX

A. ANALYZING ACCESS CONTROL

As discussed in Section 3, analyzing the scenarios that are consistent with a given security-design model requires reasoning about what is entailed by the model's permissions and associated constraints, as well as any invariants of the underlying design model.

Our current approach to analyzing access-control scenarios is based on a mapping from OCL to first-order logic [14]. In a nutshell, this mapping is defined recursively over the structure of OCL expressions. Boolean expressions are translated to formulas, mirroring their logical structure; integer expressions are basically copied. Collections are translated to predicates, whose meaning is defined by auxiliary formulas generated by the mapping. Association-ends are translated to predicates, which are also defined by auxiliary formulas. Finally, attributes are translated to uninterpreted functions and classes are translated to predicates.

Based on this mapping, we can answer questions about the consistency of a scenario satisfying a given property, with respect to an access control policy model, using satisfiability modulo theories (SMT) solvers. For example, given the access control policy model of Figure 4, consider the property of someone being able to change his own salary. We use the SMT solver Yices [19] to analyze the consistency of scenarios satisfying this property with respect to this model.

First, using our mapping, we formalize in Yices the information contained in the (underlying) data model, i.e., the model shown in Figure 3.

```
(define Employee::(-> int bool))
(define supervisedBy::(-> int int bool))
(define supervises::(-> int int bool))

; type properties of supervisedBy and supervises
(assert (forall (x::int) (forall (y::int)
  (=> (supervisedBy x y) (Employee y)))))
(assert (forall (y::int) (forall (x::int)
  (=> (supervises y x) (Employee x)))))

; multiplicity of supervisedBy
(assert (forall (x::int) (forall (y::int)
  (=> (and (Employee x) (and (Employee y)
         (supervisedBy x y)))
      (forall (z::int)
         (=> (and (Employee z)
               (supervisedBy x z))
      (= y z)))))))

; relationship between supervisedBy and supervises
  (assert (forall (x::int) (forall (y::int)
(=> (supervisedBy x y) (supervises y x)))))
(assert (forall (x::int) (forall (y::int)
  (=> (supervises y x) (supervisedBy x y)))))

; invariant: nobody is his (or her) own supervisor:
(assert (forall (x::int)
     (=> (Employee x) (not (supervisedBy x x)))))
```

Second, we also use our mapping to formalize the OCL expression stating that someone satisfies the contraint for changing his own salary, namely:

```
Employee.allInstances()->exists(self, caller|
    self.supervisedBy->includes(caller)
    and self=caller)
```

The resulting assertion in Yices is the following:

```
(assert (exists (self::int) (exists (caller::int)
   (and (Employee self) (and (Employee caller)
   (and (supervisedBy self caller)
   (and (= self caller)
```

Finally, to check that the access control policy model prevents anyone from changing his own salary, we check if all of the above assertions are satisfiable. As expected, the answer automatically provided by Yices is `unsat`.