Automatic Generation of Security-Aware GUI Models

Michael Schläpfer[1], Marina Egea[1], David Basin[1], and Manuel Clavel[2,3]

[1] ETH Zürich, Switzerland
{basin,marinae}@inf.ethz.ch,michschl@student.ethz.ch
[2] IMDEA Software Institute, Madrid, Spain
manuel.clavel@imdea.org
[3] Universidad Complutense de Madrid, Spain
clavel@sip.ucm.es

Abstract. In typical software applications, users access application data using GUI widgets. There is an important, but little explored, link between visualization and security: when the application data is protected by an access-control policy, the application GUI should be *aware of* and *respect* this policy. For example, a widget should not give users options to execute actions on the application data that they are not authorized to execute. However, GUI designers are not (and usually should not be) aware of the application data security policy. To solve this problem, we define in this paper a many-models-to-model transformation that, given a security-aware data model and a GUI model, makes the GUI model also security-aware.

1 Introduction

In typical software applications, users access application data using GUI widgets: data is created, deleted, read, and updated using text boxes, check boxes, combo boxes, buttons, and the like. There is an important, but little explored, link between visualization and security: When the application data is protected by an access-control policy, the application GUI should be *aware of* and *respect* this policy. Otherwise, users will often experience frustration. For example, after filling out a long electronic form, the user may be informed that the form cannot be submitted because she lacks permissions to execute the actions that are required on the application data. However, the GUI designers are not (and usually should not be) aware of the application data security policy. Their job is simply to design the GUI's layout and to specify its behaviour, i.e., which events will trigger which actions on which application data and/or application widgets.

To solve this problem, we define in this paper a many-models-to-model transformation that, given a security-aware data model and a GUI model, makes the GUI model also security-aware. This model transformation is the key component of our proposal for designing security-aware application GUI models. Figure 1 illustrates this proposal. The process of designing a security-aware GUI has the following parts. First, software engineers specify the application data model M.

Then, security engineers specify, in the security model $S(M)$, the application-data access-control policy, and GUI designers specify the application GUI model $G(M)$. Finally, the application security-aware GUI model $S(G(M))$ is automatically generated from the security model $S(M)$ and the GUI model $G(M)$.

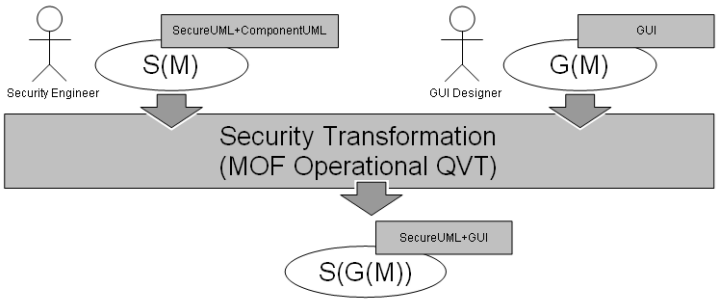

Fig. 1. Generating security-aware application GUIs.

In Section 2 we introduce the source and target models of the transformation by describing their respective metamodels (namely, SecureUML+ComponentUML, GUI, and SecureUML+GUI). Then, in Section 3, we describe the transformation as a QVT operational transformation and, in Section 4, we discuss its correctness. Finally, in Section 5, we give an overview of the planned extensions of the ideas presented in this paper. We illustrate our ideas on a running example, namely, the design of a simple GUI for a phone-book application. As part of our work, we have implemented the transformation using the Operational QVT transformation engine that is provided within the M2M Project, a subproject of the Eclipse Modeling Framework. Our tool is available at [14] along with documentation and examples.

Our model-transformation based approach for designing security-aware GUI models has three principle advantages over traditional software development approaches.

1. Security engineers and GUI designers can independently model what they know best (or know at all).
2. Security engineers and GUI designers can independently change their models and these changes are automatically propagated to the final security-aware GUI models.
3. GUI designers, even if they do not know the underlying security policy, can still check its impact on their designs. They can use the final security-aware GUI models to check that they are designing the right GUI to give the (authorized) users access to the (intended) application data.

Our proposal for automatically generating security-aware GUI models is the corner stone of a more ambitious project for making model-driven security an

effective and useful approach for generating multiple system layers as part of a security-intensive industrial software development. A crucial property of these systems that we directly pursue with our model-transformation based approach, is *conformance*. For the case addressed in this paper, conformance means that executing events on the GUI layer never leads to program exceptions from the access-control security policy implemented at the persistent layer.

2 The Transformation Model Types

Model transformation is the process of converting some models $M_1, \ldots, M_n$ (the transformation *source* models) into other models $M'_1, \ldots, M'_m$ (the transformation *target* models). In this section, we define the *types* of the source and target models of our model transformation by introducing their respective metamodels: namely, the SecureUML+ComponentUML metamodel and the GUI metamodel (which define our source models' types), and the SecureUML+GUI metamodel (which defines our target model's type). Since SecureUML+ComponentUML and SecureUML+GUI are both dialects of SecureUML, we begin this section by briefly introducing SecureUML.

The SecureUML language. SecureUML is a language based on RBAC [6] for modeling access-control policies on protected resources [2]. The policies that can be specified in SecureUML are of two kinds: those that depend on static information, namely the assignments of users and permissions to roles; and those that depend on dynamic information, namely the satisfaction of authorization constraints in the current system state. However, SecureUML leaves open what the protected resources are and which actions they offer to clients. These are specified in a so-called *dialect* and depend on the primitives for constructing models in each dialect's system-design modeling language. Figure 2 shows the SecureUML metamodel: each SecureUML dialect will basically declare its own protected resources as subclasses of *Resources* and the actions that they offer to clients as subclasses of *Atomic* or *Composite Actions*.

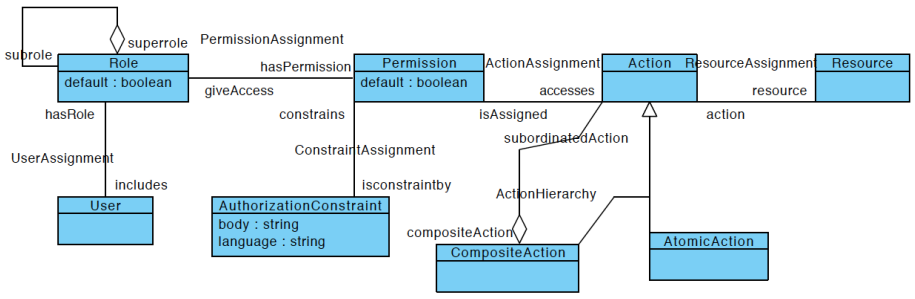

Fig. 2. The SecureUML metamodel.

2.1 The source model types

The source models of our many-models-to-model transformation are SecureUML+ComponentUML models and GUI models. In a nutshell, the former specify the policies for accessing the application data, while the latter specify which of the actions on the application data are triggered by which application GUI events. Importantly, our source models have the same underlying application-data model.

The SecureUML+ComponentUML language. This language combines SecureUML with ComponentUML, which is a simple language for modeling component-based systems. ComponentUML provides a subset of UML class models: *Entities* can be related by *Associations* and may have *Attributes* and *Methods*. The SecureUML+ComponentUML metamodel, partially shown in Figure 3, provides the connection between SecureUML and ComponentUML. It specifies the following.

- The protected resources, namely, *Entities*, as well as their *Attributes*, *Methods*, and *AssociationEnds* (but not *Associations* as such).
- The actions on these protected resources and their hierarchies. These are shown in the following table.

Resource	Actions
Entity	create, read, update, delete, full access
Attribute	read, update, full access
Method	execute
Association end	read, update, full access

In this table, composite actions are underlined. They are used to group primitive actions into a hierarchy of higher-level ones: e.g., full access on an attribute includes both read and update access on this attribute, and full access on an entity includes both full access on the entity attributes, entity methods, and methods for entity creation and deletion.

In [2] a UML profile is defined for drawing SecureUML+ComponentUML models, which we summarize here. A role is represented by a UML class with the stereotype $\langle\langle Role \rangle\rangle$ and an inheritance relationship between two roles is defined using a UML generalization relationship. The role referenced by the arrowhead of the generalization relationship is considered to be the superrole of the role referenced by the tail. A permission, along with its relations to roles and actions, is defined in a single UML model element, namely an association class with the stereotype $\langle\langle Permission \rangle\rangle$. The association class connects a role with a UML class representing a protected resource, which is designated as the root resource of the permission. The actions that such a permission refers to may be actions on the root resource or on subresources of the root resource. Each attribute of the association class represents the assignment of an action to the permission, where the action is identified by the name and the type of the attribute. ComponentUML entities are represented by UML classes with

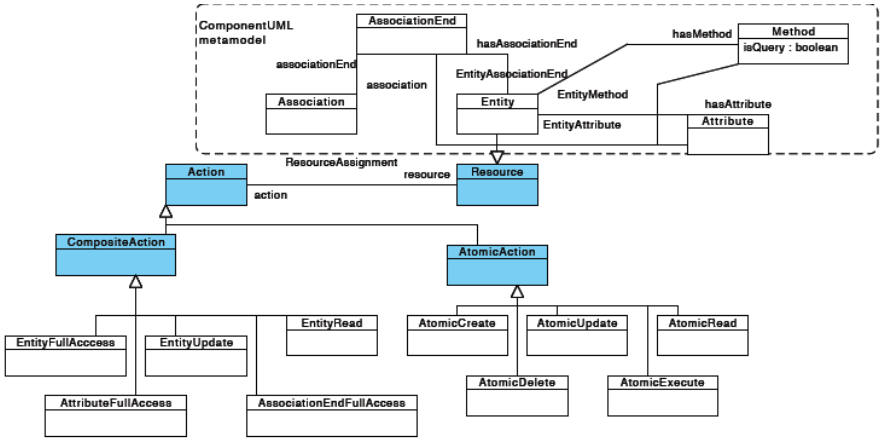

Fig. 3. The SecureUML+ComponentUML metamodel (partial).

the stereotype $\langle\langle Entity \rangle\rangle$. Every method, attribute, or association end owned by such a class is automatically considered to be a method, attribute, or association end of the entity.

Example 1. Consider a basic phone-book application, called PhoneBook. Each entry in the underlying phone directory consists of a name and a phone number. The access to this data is controlled by the following policy:

- Users are only allowed to read people's name and phone numbers.
- Supervisors are allowed to read people's name and phone numbers, as well as to change phone numbers.
- Administrators are allowed to create and delete entries in the phone directory, as well as to read and write them.

Figure 4 shows the SecureUML+ComponentUML model that specifies the above policy.

The GUI language. We now introduce a simple language for modeling GUIs, which is however rich enough for our present purposes. Its metamodel is shown in Figure 5. Application GUIs consist of *Widgets* that are displayed inside *Containers*, which are themselves *Widgets*. Each widget has a (possibly empty) set of *ActionEvents* associated to it, which specifies how the widget reacts to events. For the purpose of this paper, we can assume that each event can only trigger one action on the application data.[4] We also assume that, in each instance of *ActionEvent*, the value of the attribute *modelAction* is a SecureUML+ComponentUML atomic action and that this action has as its root resource an entity declared in the ComponentUML model that specifies the underlying application-data model.

[4] In general, one event can trigger many actions, some of them acting on the application data (create, delete, read, update, or execute) and some of them acting on the GUI widgets (close, open, hide, and so on).

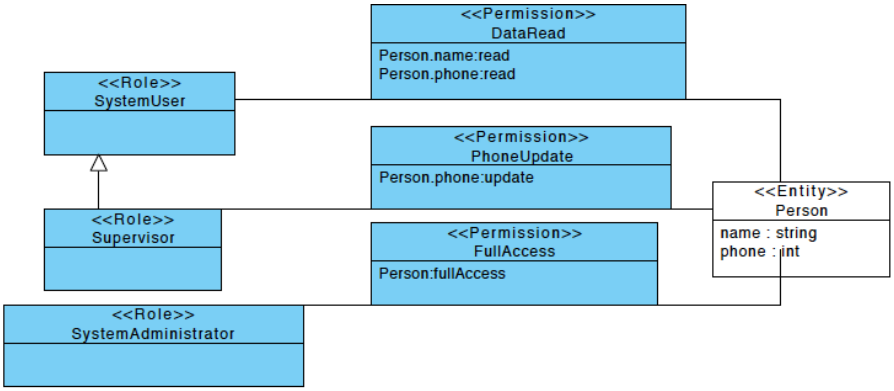

Fig. 4. A simple security policy for a PhoneBook application.

Example 2. Suppose that our PhoneBook GUI designer decides that the application should provide a basic interface for editing entries in the directory. She therefore designs a GUI consisting of a window *PhoneBook Editor*, with two entry boxes: *Name* and *Phone Number*. She also decides that, at the time of creation, each instance of *PhoneBook Editor* is associated to an instance P of the entity *Person*. Moreover, the GUI should offer the following functionality.

- *On entering* the entry box *Name*, the box should display the name of the object P, i.e., she associates the event *onEnter* with a *read* action on the attribute *name* of an instance of the entity *Person*. Similar actions should occur when entering the entry box *Phone Number*.
- *On leaving* the entry box *Name*, the text currently displayed in this box should be used to update the attribute *Name* of the person P, i.e., she associates the event *onLeave* with an *update* action on the attribute *name* of an instance of the entity *Person*. Similar actions should occur when leaving the entry box *Phone Number*.

Figure 6 shows the instance of the GUI metamodel that corresponds to the GUI model specifying the above GUI.

2.2 The target model type

SecureUML+GUI models are the target models of our many-models-to-model transformation. SecureUML+GUI combines SecureUML with the GUI modeling language introduced in the previous section. In a nutshell, SecureUML+GUI models specify who can execute which events on which widgets. The SecureUML+GUI metamodel, (partially) shown in Figure 7, provides the connection between SecureUML and GUI. It specifies the following.

- The protected resources, namely, *ActionEvents*.

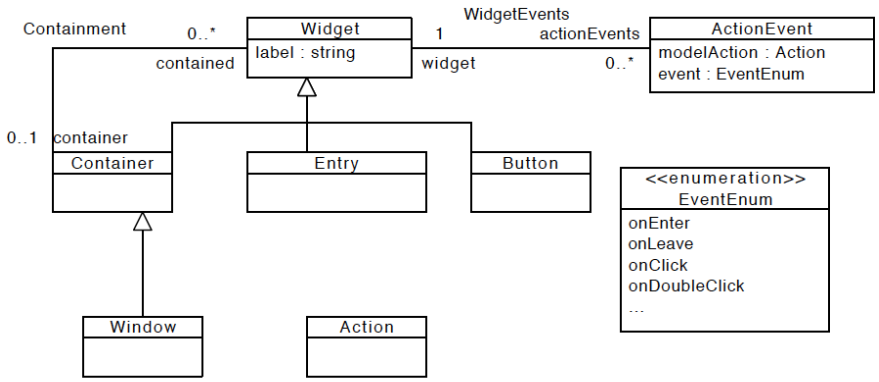

Fig. 5. A simple metamodel for GUIs.

– The actions on these protected resources, namely, *AtomicExecute*.

In the next section we will show the SecureUML+GUI model that results from applying our transformation to the SecureUML+ComponentUML model and the GUI model discussed, respectively, in Examples 1 and 2.

3 The Transformation Description

We are ready to describe a many-models-to-model transformation that automatically generates SecureUML+GUI models from SecureUML+ComponentUML models and GUI models. We assume here, as explained above, that the source models have the same ComponentUML application-data model. As we will discuss in the next section, our transformation satisfies the expected property, namely, that the (target) SecureUML+GUI model *preserves* the security policy specified in the (source) SecureUML+ComponentUML model.

We now introduce our many-models-to-model transformation as a QVT operational transformation [10, Section 8.4.6] using Operational QVT syntax. In Figure 8 we show the heading of this operational transformation. The metamodels `GUI`, `SECUMLANDCOMPUML`, and `SECUMLADGUI` are the GUI metamodel, the SecureUML+ComponentUML metamodel, and the SecureUML+GUI metamodel that were introduced in Section 2; their definitions are available at [14]. The operational transformation is defined by mapping functions, which are executed sequentially. Due to space limitations, we can only describe these functions here; their full definitions are also available at [14]. The final target models are obtained in two steps.

Step 1: The model elements of the target model are created as follows.

– The *Roles* in the (source) SecureUML+ComponentUML model are copied, along with their hierarchies, in the (target) SecureUML+GUI model, using the following mapping functions.

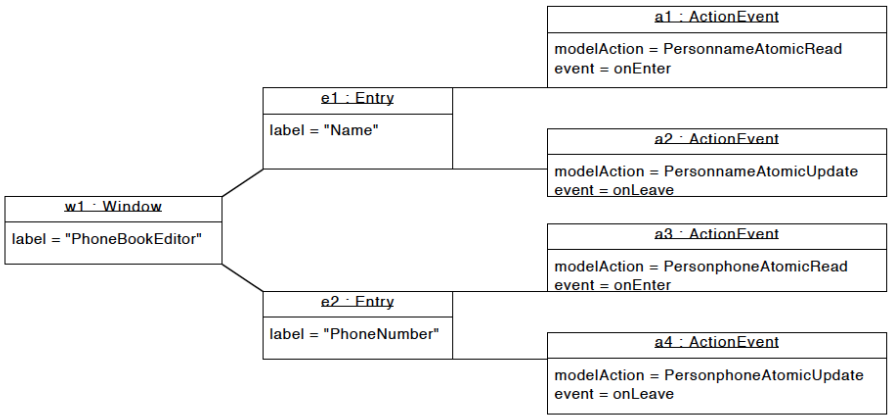

Fig. 6. A simple GUI for editing PhoneBook entries.

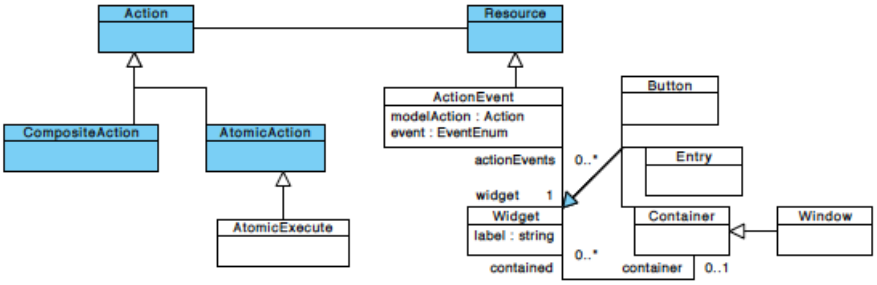

Fig. 7. The SecureUML+GUI metamodel (partial).

```
secPolicy.objects()[SECUMLANDCOMPUML::Role]->map Role_to_Role();
secPolicy.objects()[SECUMLANDCOMPUML::Role]->map
              preserve_Role_hierarchy();
```

– The *Widgets* in the (source) GUI model are copied, along with their containment relationships and their associated *ActionEvents*, in the (target) SecureUML+GUI model, using the following mapping functions.

```
guiModel.objects()[GUI::ActionEvent]->map
             ActionEvent_to_ActionEvent();
guiModel.objects()[GUI::Window]->map Widget_to_Widget();
guiModel.objects()[GUI::Entry]->map Widget_to_Widget();
guiModel.objects()[GUI::Button]->map Widget_to_Widget();
guiModel.objects()[GUI::Widget]->map
             preserve_containment_hierarchy();
```

– For each *ActionEvent* in the (source) GUI model, an *AtomicExecute* action is created and linked to the *ActionEvent* as to its root resource in the (target) SecureUML+GUI model using the following mapping function.

```
guiSecPolicy.objects()[SECUMLANDGUI::ActionEvent]->map
                addAtomicExecuteAction();
```

Step 2: The permission assignment in the target model are created as follows.

– For each *ActionEvent*'s *action* in the (source) GUI model, and for each *Role* that is allowed to perform this *action* in the (source) SecureUML+ComponentUML model, a *Permission* is created in the (target) SecureUML+GUI model that grants access to *Role* to execute the *ActionEvent*'s *event*. This is accomplished using the following mapping function.

```
guiSecPolicy.objects()[SECUMLANDGUI::ActionEvent]->liftPermissions();
```

```
modeltype GUI uses "http://gui/1.0";
modeltype SECUMLANDCOMPUML uses "http://secumlandcompuml/1.0";
modeltype SECUMLANDGUI uses "http://secumlandgui/1.0";

transformation SecurityTransformation(in guiModel : GUI,
                in secPolicy : SECUMLANDCOMPUML,
                out guiSecPolicy : SECUMLANDGUI);

main() {
  /* Step 1: Creating the model elements of the target model */
  secPolicy.objects()[SECUMLANDCOMPUML::Role]->map Role_to_Role();
  secPolicy.objects()[SECUMLANDCOMPUML::Role]->map
                preserve_Role_hierarchy();
  guiModel.objects()[GUI::ActionEvent]->map
                ActionEvent_to_ActionEvent();
  guiModel.objects()[GUI::Window]->map Widget_to_Widget();
  guiModel.objects()[GUI::Entry]->map Widget_to_Widget();
  guiModel.objects()[GUI::Button]->map Widget_to_Widget();
  guiModel.objects()[GUI::Widget]->map
                preserve_containment_hierarchy();
  guiSecPolicy.objects()[SECUMLANDGUI::ActionEvent]->map
                addAtomicExecuteAction();

  /* Step 2: Creating permission assignments in the target model */
  guiSecPolicy.objects()[SECUMLANDGUI::ActionEvent]->liftPermissions();
```

Fig. 8. The many-models-to-model transformation's heading in QVTO syntax.

Example 3. We show in Figure 9 the (relevant aspects of the) SecureUML+GUI model that results from applying our many-models-to-model transformation to the SecureUML+ComponentUML model and the GUI model discussed, respectively, in Examples 1 and 2. Here, to draw our resulting SecureUML+GUI model, we use a UML profile similar to the one available for SecureUML+ComponentUML models, except that the stereotype $\langle\langle Entity \rangle\rangle$ is now reserved for *Widgets*, which contains as methods their associated *ActionEvents' events*.

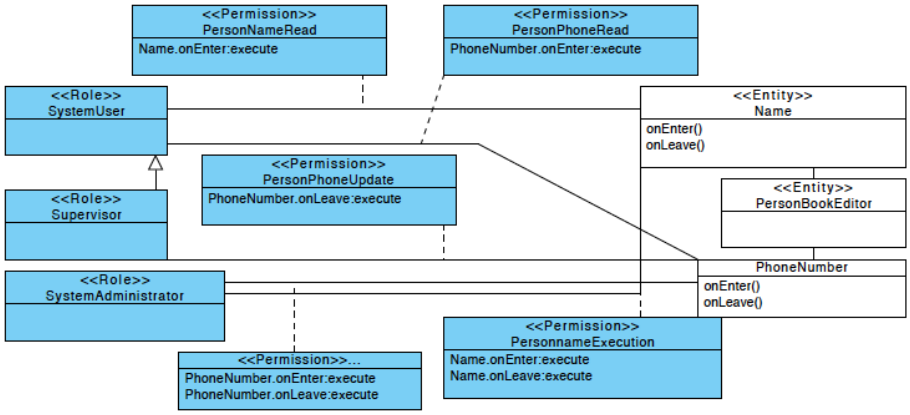

Fig. 9. A security-aware simple GUI for editing PhoneBook entries.

Interestingly, a simple analysis of the resulting SecureUML+GUI model in Example 3 reveals that only *Administrators* should be allowed to open a *PhoneBook Editor* window, since only they can execute *all* the events associated to the widgets contained in a *PhoneBook Editor* window (in [1] it is explained how SecureUML models can be automatically analysed using OCL queries). Of course, this information is crucial for the GUI designer in order to validate their GUIs, i.e., to check that she is designing the right graphical interface to give the (authorized) users access to the (intended) application data. She may realize, for example, that, in order to give *Supervisors* access to editing peoples' phone number, another GUI is needed with no action associated this time to the event of leaving the *Name* entry box,

4 The correctness of the transformation

In this section, we discuss the correctness of our transformation. Our claim is that the (target) SecureUML+GUI model $S(G(M))$ *preserves* the security policy specified in the (source) SecureUML+ComponentUML model $S(M)$. More specifically, we claim that a role is allowed to execute an event in the $S(G(M))$

model only if it is allowed to execute the action that is associated to this event in the $S(M)$ model.

In [1] we proposed a metamodel-based approach for automatically analyzing security-design models in a semantically precise and meaningful way. More concretely, we showed that security properties of security-design models can be expressed as formulas in OCL [**?**], the Object Constraint Language of UML. We can formalize queries about the relationships between users, roles, permissions, and actions, and we can answer such queries by evaluating them on the instances of the SecureUML metamodel that represent the security-design model under consideration. We also defined in [1] a number of OCL operators that formalize different aspects of the access-control information contained in the security-design models. In particular, we defined in [1] an operator allAtomics() that, given a role, returns all the atomic actions that a user in this role can perform:

---

context Role::allAtomics():Set(AtomicAction) **body**:
self.allPermissions().allActions()−>asSet()

The operator allPermissions returns the collection of permissions (directly or indirectly) assigned to a role. The operator allActions returns the collection of atomic actions whose access is (directly or indirectly) granted by a permission. We can now use the operator allAtomics() to formally state the correctness property of transformation as follows.

Remark 1. Given a SecureUML+ComponentUML model $S(M)$ and a GUI model $G(M)$, the SecureUML+GUI model $S(G(M))$ resulting from our transformation satisfies the following property: Let *rl* be a *Role* in $S(M)$ and let *acev* be an *ActionEvent* in $G(M)$. Let *ac* be the *Action*-value of *acev*'s attribute *modelAction* in $G(M)$. Suppose now that there exists an *AtomicExecute* action *acex* in $S(G(M))$ that is linked to the *ActionEvent* *acev* as its root resource. Then, *rl*.allAtomicActions()−>includes(*acex*) evaluates to true in $S(G(M))$ only if *rl*.allAtomicActions()−>includes(*ac*) evaluates to true in $S(M)$.

The above remark follows from the fact that: i) by Step 1 of our transformation, the role hierarchy in $S(G(M))$ is *exactly* the one specified in $S(M)$, and ii) by Step 2 of our transformation, a role is granted permission to execute an action-event in $S(G(M))$ *only if* this role is granted permission to execute in $S(M)$ the action associated to this action-event (as the value of its attribute *modelAction*).

5 The Transformation Extensions

We are currently extending the work presented here in various, related directions. A first direction is to consider application-data security policies that also depend on dynamic information, namely the satisfaction of authorization constraints in the current system state. In this context, our transformation function

must include functions that correctly map authorization constraints in the security models to authorization constraints in the security-aware GUI models. A second direction is to apply our approach to more realistic GUI designs. Here, we will work with GUI models that associate multiple actions, both on the application data and on the GUI widgets, to single events. In this context, Step 2 in our transformation function is less direct. In particular, there could be events whose execution can not be granted to anybody. This would happen if no one is allowed to execute *all* the actions associated to this event. A third direction is to generate GUI models that are not only security-aware, but also *smart*. Smartness is relevant since events can also trigger actions on GUI widgets. For example, in a smart security-aware GUI, widgets should not give users the option to open other widgets when these widgets in turn give them options to execute actions on the application data that they are not authorized to execute. Making security-aware GUIs also smart requires "lifting" permissions in two directions: i) from the widgets whose events triggered actions on application data to the widgets whose events triggered actions on those widgets and ii) from the widgets that are contained in other widgets to the widgets that contain those widgets.

Overall, we aim to provide GUI designers with better models and tools for building and analyzing GUIs for security-critical applications, including tools for automatically checking that, using a given GUI, (authorized) users can indeed access the (intended) application data, or tools for automatically building GUIs intended for specific users, in which all (and only) the (authorized) actions on the application data are indeed accessible by the (intended) users.

6 Related Work

Creating user interfaces is a common task in application development. It can also be very time consuming and therefore expensive. Many proposals have been made, and tools have been built, that aim to reduce the efforts required to build effective and user-friendly graphical interfaces. Surprisingly, despite all these initiatives, until now there has been no research into the systematic design of GUIs whose functionality should adhere to the security policy designed for the underlying application-data model.

In the modeling community, other researchers have investigated how to extend existing modeling environments for GUI modeling. For instance, [5, 3, 4] propose various UML extensions for this purpose. There are also approaches [8] suggesting the use of off-the-shelf web widget libraries to develop web-based user interfaces for semantic web applications, where developers can use RDF constructs to map the data contained in the underlying data model to the model implemented by the widget. More directly related to MDA, [12] reviews the tools that currently support general modeling, model transformations, model weaving, and model constraints in relation to the special needs of the human-computer interaction (HCI) community. Another survey is given in [13], which focuses on transformation tools for model-based user interface development. In relation to security, [15, 16] uses QVT to handle security requirements in an

MDA setting; in particular, to obtain the secure logical scheme from conceptual models. Also, [7] uses the Sectet-framework to integrate security requirements with models at the abstract level and proposes a QVT-based chain of tools that transform these models into artefact's configuring security components of a Web services-based architecture. To the best of our knowledge, none of these approaches is appropriate for modeling application-data access-control security policies at the GUI level. Also, we are not aware of other approaches based on model-transformations for automatically generating security-aware GUI models from security-design models, that is, from models that integrate system designs with their access-control policy.

In the programming community, independent of model-driven initiatives, numerous projects have addressed implementing graphical user interfaces for application data. For example, [9] proposes enriching the application code source with annotations that control the generation of the graphical user interfaces. Other researchers have designed and implemented specialized tools that support the automatic generation of graphical user interfaces meeting their own specific requirements. These tools simplify configuring personal services, enabling the combination of different kinds of events [11]. Also, there are many GUI builders, either integrated in IDEs or available as plug-ins, that simplify the task of creating application GUIs in different programming languages.

7 Conclusions

In this paper we have presented an approach based on model-transformation for automatically generating security-aware GUI models. Given a security-aware data model and a GUI model, our transformation makes the GUI model also security-aware. We have introduced the source and target models of this transformation (by describing their respective metamodels) and we have described the main mapping functions that define the transformation as a QVT operational transformation. We have also discussed the correctness of our transformation. As part of our work, we have implemented this approach using the Operational QVT transformation engine that is provided within the M2M Project, a subproject of the Eclipse Modeling Framework.

Our model-transformation based approach for designing security-aware GUI models has three main advantages over traditional software development approaches. First, security engineers and GUI designers can independently model what they know best. Second, security engineers and GUI designers can independently change their models, and these changes are automatically propagated to the security-aware GUI models. Third, GUI designers can use the security-aware GUI models to check that they are designing the right GUI to give the authorized users access to the intended application data. The transformation presented here is the corner stone of a more ambitious project for making model-driven security an effective and useful approach for generating multiple system layers as part of a security-intensive industrial software development.

References

1. D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. To appear in *Information and Software Technology Journal*, Elsevier. Special issue on *Model Based Development for Secure Information Systems*, 2008.
2. 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.
3. K. Blankenhorn and W. Walter. Extending UML to GUI modeling. `http://www.bitfolge.de/pubs/MC2004_Poster_Blankenhorn.pdf`, 2004.
4. P. Pinheiro da Silva and N. W. Paton. UMLi: The unified modeling language for interactive applications. In *UML 2000 - The Unified Modeling Language. Advancing the standard. Third International Conference*, pages 117–132. Springer, 2000. `http://trust.utep.edu/umli/`.
5. TATA Research Development and Design Center. Heavyweight extension of UML for GUI modeling : A template based approach. `http://www.omg.org/news/meetings/workshops/presentations/uml2001_presentations/10-2_Venkatesh_typesasStereotypes.pdf`, 2001.
6. D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for Role-Based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.
7. M. Hafner, M. Alam, and R. Breu. Towards a MOF/QVT-Based domain architecture for Model Driven Security. In *Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 275–290. Springer Berlin / Heidelberg, 2006.
8. M. Hildebrand and J. van Ossenbruggen. Configuring semantic web interfaces by data mapping. In Siegfried Handschuh, Tom Heath, and Vinhtuan Thai, editors, *Visual Interfaces to the Social and the Semantic Web (VISSW 2009)*, volume 443, February 2009.
9. J. Jelinek and P. Slavik. GUI generation from annotated source code. In *TAMODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams*, pages 129–136, New York, NY, USA, 2004. ACM.
10. Object Management Group. MOF-Queries, Views and Transformations (QVT)-Final adopted specification. Technical report, OMG, 2005. `www.omg.org/docs/ptc/05-11-01.pdf`.
11. M. Ogura, H. Mineno, N. Ishikaw, T. Osano, and T. Mizuno. *Automatic GUI Generation for Meta-data Based PUCC Sensor Gateway*, volume 5179 of *LNCS*, pages 159–166. Springer Berlin–Heidelberg, 2008.
12. J.L. Pérez-Medina, S.Dupuy-Chessa, and A.Front. A survey of model driven engineering tools for user interface design. In *Task Models and Diagrams for User Interface Design*, volume 4849 of *LNCS*, pages 84–97. Springer Berlin / Heidelberg, 2007.
13. R.Schaefer. A survey on transformation tools for model based user interface development. In *Human-Computer Interaction. Interaction Design and Usability.*, volume 4550 of *LNCS*, pages 1178–1187. Springer Berlin / Heidelberg, 2007.
14. M. Schläpfer. A tool for generating security-aware GUI models. `http://n.ethz.ch/student/michschl`, 2009.
15. E. Soler, J. Trujillo, E. Fernandez-Medina, and M. Piattini. Application of QVT for the development of secure data warehouses: A case study. In *The Second International Conference on Availability, Reliability and Security, 2007. ARES 2007.*, 2007.

16. E. Soler, J. Trujillo, E. Fernandez-Medina, and M. Piattini. A set of QVT relations to transform PIM to PSM in the design of secure data warehouses. In *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 644–654, Washington, DC, USA, 2007. IEEE Computer Society.