Automatic Generation of Smart, Security-Aware GUI Models

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

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

Abstract. In many software applications, users access application data using graphical user interfaces (GUIs). There is an important, but little explored, link between visualization and security: when the application data is protected by an access control policy, the GUI should be aware of this and respect the policy. For example, the GUI should not display options to users for actions that they are not authorized to execute on application data. Taking this idea one step further, the application GUI should not just be security-aware, it should also be smart. For example, the GUI should not display options to users for opening other widgets when these widgets will only display options for actions that the users are not authorized to execute on application data. We establish this link between visualization and security using a model-driven development approach. Namely, we define and implement a many-models-to-model transformation that, given a security-design model and a GUI model, makes the GUI model both security-aware and smart.

1 Introduction

In many programs, users access application data using GUI widgets: data is created, deleted, read, and updated using text boxes, check 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. For example, the GUI should not display options to users for actions that they are not authorized to execute on application data. This prevents user frustration, for example, from filling out a long electronic form only to have the server reject it because the user lacks a permission to execute some associated action on the application data. Taking this idea one step further, the GUI should not, for example, display options to users to open other widgets when these widgets only display options for actions that the users are not authorized to execute on application data. That is, the application GUI should not just be security-aware but also *smart*.

Visualization and security

To see how this link between GUIs and security policies might look, consider the following example: an application for managing employee information. This information will include, among other data, employees' names, phone numbers, and salaries. Suppose now that the employee information is protected by an access control policy that includes, among other clauses, the following:

– All users can read employees' names.
– Only administrators and supervisors can read and update employees' phone numbers.
– Only supervisors can read employees' salaries.

Suppose now that, as shown in Figure 1, our application GUI includes the following windows:

Window #1. This is our main window. Here, users can edit employee data by clicking on the Edit Phone Number-button.

Window #2. Users can select an employee from a list of names, shown in a combo box, and view the information associated to the selected employee by clicking on the View-button.

Window #3. Users can read in the Name, Phone, and Salary-labels, respectively, the name, phone number, and salary of the selected employee. Moreover, users can edit the phone number by clicking on the Edit-button.

Window #4. Users can update the phone number of the selected employee by typing the new number into the Phone-entry and clicking on the OK-button.

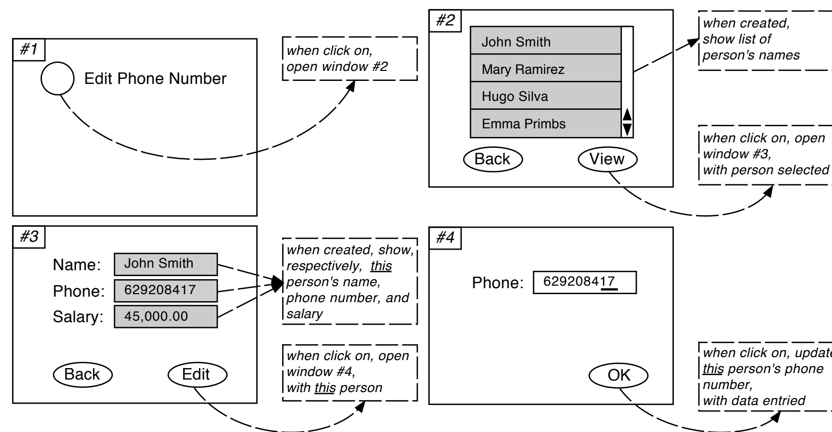

Fig. 1. A simple GUI for editing employees' phone numbers

What behaviour should our GUI have if it is to be considered "security-aware"? Suppose that a user with the role administrator wants to edit an employee's phone number using our GUI. Since administrators are not authorized

to read employees' salaries, when opening Window #3, our GUI should prevent an administrator from reading this information in the Salary-label. Furthermore, how should our GUI behave if it is also to be considered "smart"? Suppose now that a user with no special privileges wants to edit an employee's phone number using our GUI. Since ordinary users are not authorized to read or update employees' phone numbers, our GUI should prevent the user from opening Window #4 by clicking on the Edit-button, since the user should not be able to do anything within this window (i.e., neither read the phone number of the selected employee in the Phone-label nor click on the OK-button to update this information).

The problem we address here is how to establish this link between visualization and security. The default, "ad-hoc" solution, namely, directly hardcoding the security policy within the GUI, is clearly inadequate. First, the GUI designer is often not aware of the application data security policy. Second, even if the designer is aware of it, hardcoding the application data security policy within the GUI code is cumbersome and error-prone, if done manually. Finally, any changes in the security policy will require manual changes to the GUI code where this policy is hardcoded, which again is a cumbersome and error-prone task.

Our approach: model-transformation

We propose in this paper a model-driven approach that links visualization and security. The key idea is that this link 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 graphical user interface. The key component of our proposal is a many-models-to-model transformation which, given a security-design model (specifying the access control policy on the application data) and a GUI model (specifying the actions triggered by the events supported by the application's graphical interface), automatically generates a GUI model that is both security-aware and smart. Thus, under our proposal, illustrated in Figure 2, the process of modeling a smart, security-aware GUI has the following parts.

1. Software engineers specify the application-data model $\mathcal{C}$.
2. Security engineers specify the security-design model $\mathcal{S}_{\mathcal{C}}$.
3. GUI designers specify the application GUI model $\mathcal{G}_{\mathcal{C}}$.
4. A many-models-to-model transformation automatically generates a smart, security-aware GUI model $\mathcal{M}_{(\mathcal{G}_{\mathcal{C}},\mathcal{S}_{\mathcal{C}})}$ from the security model $\mathcal{S}_{\mathcal{C}}$ and the GUI model $\mathcal{G}_{\mathcal{C}}$.

To show the applicability of our proposal, we have implemented an Eclipse-based application that automatically generates smart, security-aware GUI models from security-design models and GUI models [7]. Moreover, it automatically generates smart, security-aware web applications from the generated smart, security-aware GUI models. Specifically, our Eclipse-application includes the following parts.

– A GMF editor for drawing application-data models.

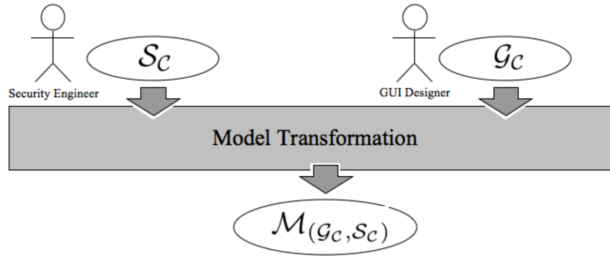

Fig. 2. Modeling a smart and security-aware GUI.

– A plugin for generating user-friendly GMF editors for drawing security-design models and GUI models.
– A plugin for generating smart, security-aware GUI models from security-design models and GUI models.
– A plugin for generating web applications from smart, security-aware GUI models.

Due to space limitations, we only report on our plugin for generating smart, security-aware GUI models. This plugin implements our many-models-to-model transformation as a QVT operational transformation.

Applicability and extensions

The applicability of our approach crucially depends on the expressiveness of the modeling languages used to specify the access control policies and the graphical user interfaces. In this paper, however, we focus on the two main ideas behind our approach.

1. The link between visualization and security is essentially given by the data actions, since they are both controlled by the security policy and triggered by the events supported by the graphical user interface.
2. This link can be systematically established using an appropriate many-models-to-model transformation to generate smart, security-aware GUI models from the models specifying the security policy and the models specifying the graphical user interfaces.

To explain these ideas, we present our approach using abstract notions of both security-design models and (smart, security-aware) graphical user interface models. For the sake of illustration, we will also use concrete modeling languages, which provide the source and target models of a many-models-to-model transformation that we will introduce to exemplify our approach; however, our approach is not restricted to or dependant on the use of these languages or our particular many-models-to-model transformation. In fact, our Eclipse-based application for generating smart, security-aware GUIs [7] currently supports a modeling language for specifying graphical user interfaces that is significantly more expressive

than the one introduced in this paper. For example, it allows one to associate data to widgets, to pass information from one widget to another widget, to jump from one widget to another widget, and to call actions on data with parameters. Notice that some of these features are, for example, needed to fully modeled the graphical user interface described in Figure 1.

Organization. In Sections 2 and 3 we introduce security-design models and GUI models. Afterwards, in Sections 4 and 5, we introduce smart, security-aware GUI models and we define a many-models-to-model transformation that automatically generates smart, security-aware GUI models from security-design models and GUI models. We conclude with a discussion of related and future work. Throughout the paper we will use the employee information system example, given above, as our running example.

2 Security-design models

Model-driven security (MDS) [2] is a specialization of model-driven development for developing secure systems. In this approach, designers specify system models along with their security requirements and use tools to automatically generate system architectures from the models, including complete, configured access control infrastructures. MDS is centered around the construction (and analysis) of *security-design models*, which are models that combine security requirements with system designs.

In this section, we define security-design models as they will be considered throughout this paper. Our focus is on access control security requirements. We first provide an abstract definition (Definition 2) of security-design models, independent of any modeling language that may be used for specifying them. Then, we introduce a specific language (SecureUML+ComponentUML) for modeling security-design models.

We begin by defining system design models (Definition 1) and by introducing a specific modeling language for them (ComponentUML). For the sake of simplicity, we will consider that system designs are component-based, and we will use the following (rather simple) notion of a component-based design model throughout this paper

Definition 1. *A* component-based design model $\mathcal{C}$ *is a 4-tuple*

$$\mathcal{C} = \langle E, At, As, Md \rangle$$

that specifies the entities E which play a role in the system as well as their properties, given by their attributes At, associations-ends As, and their methods Md.

Example 1. The component-based design model specifying the data model underlying our running example will consist of a single entity (*Person*), with three attributes (*name*, *phone number*, and *salary*).

The ComponentUML language. 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. Its metamodel is shown in Figure 3 (inner rectangle).

Each valid instance of the ComponentUML metamodel specifies a component-based design model $\langle E, At, As, Md\rangle$ whose components are defined by the following OCL expressions:

$E = \mathsf{Entity.allInstances()}$.
$At = \mathsf{Attribute.allInstances()}$.
$As = \mathsf{AssociationEnd.allInstances()}$.
$Md = \mathsf{Method.allInstances()}$.

We are now ready to define security-design models.

Definition 2. *Let $\mathcal{C}$ be a component-based model $\mathcal{C} = \langle E, At, As, Md\rangle$. Then, a* security-design model *$\mathcal{S}_\mathcal{C}$ for $\mathcal{C}$ is a 5-tuple*

$$\mathcal{S}_\mathcal{C} = \langle Rs, DaAc, Rl, RsDaAc, DaAu\rangle,$$

with $Rs = (E \cup At \cup As \cup Md)$, $RsDaAc : Rs \longrightarrow \mathcal{P}(DaAc)$, and $DaAu : DaAc \longrightarrow \mathcal{P}(Rl)$. The model $\mathcal{S}_\mathcal{C}$ specifies a security policy for accessing the resources (namely, the entities and their properties) in the component-based system modeled by $\mathcal{C}$. More concretely, $\mathcal{S}_\mathcal{C}$ specifies

- *the actions $DaAc$ whose access policy is modeled;*
- *the specific actions $RsDaAc(rs)$ supported by a given resource $rs \in Rs$;*
- *the roles Rl that users may adopt when interacting with the system; and*
- *the roles $DaAu(daac) \subseteq Rl$ that are authorized to execute a given action $daac \in DaAc$.*

Example 2. The security-design model specifying the security policy of our running example will define, for example, that:

- the roles that users may adopt when using the system are *all* (for users with no special privileges), *administrator*, and *supervisor*;
- the actions supported by the resources include, for example, to *read* and *update* the *phone number* resource; and
- the role *supervisor*, for example, is authorized to execute a *read* action on (any) *salary* resource, but not the other two roles.

The SecureUML language. This is a modeling language based on RBAC [5] for formalizing 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. 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 the associated system-design modeling language. Each SecureUML dialect basically declares its own protected resources and the actions that they offer to clients.[4]

The SecureUML+ComponentUML language. This is a SecureUML dialect that connects SecureUML with ComponentUML, providing a convenient language for specifying security-design models. Its metamodel is shown in Figure 3.

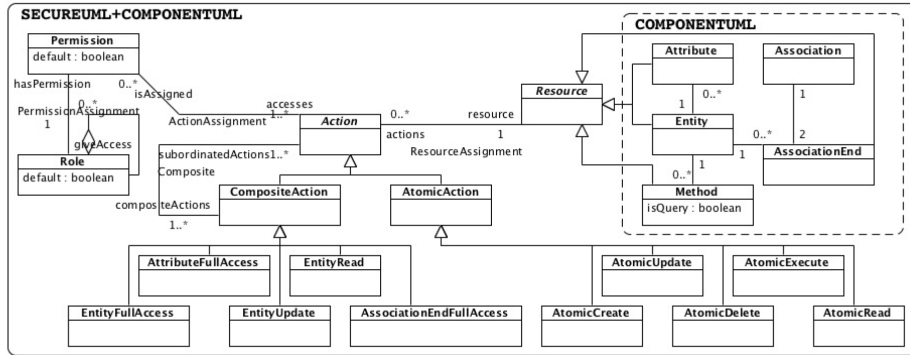

Fig. 3. SecureUML+ComponentUML metamodel.

The protected resources are the entities, as well as their attributes, methods, and association-ends. The atomic actions that are offered to clients are create, delete, update, read, and execute the entity's properties or methods. The dialect also provides composite actions, which are used to group primitive actions into a hierarchy of higher-level ones. The composite actions that are offered to clients are read, update, and full access either on entities or entity's properties: e.g., full access on an attribute includes both read and update access on this attribute.

Each valid instance[5] of the SecureUML+ComponentUML metamodel specifies a security-design model $\mathcal{S}_{\mathcal{C}} = \langle Rs, DaAc, Rl, RsDaAc, DaAu \rangle$, whose components are defined by the following OCL expressions:

Rs = Resource.allInstances().
$DaAc$ = AtomicAction.allInstances().
Rl = Role.allInstances().
$RsDaAc(rs)$ = rs.actions−>select(a|a.oclIsTypeOf(AtomicAction)).
$DaAu(daac)$ = $daac$.allAssignedRoles().

[4] SecureUML also supports authorization constraints, which are assertions that restrict authorizations and are translated to run-time constraints. For the sake of simplicity, we do not consider such constraints here.

[5] We refer to [1] for the complete list of OCL invariants associated with the SecureUML+ComponentUML metamodel.

where the operation allAssignedRoles() is defined as follows:[6]

```
context AtomicAction::allAssignedRoles():Set(Roles)
body: self.compactionPlus().isassigned.allRoles()->asSet()
```

Example 3. Suppose that EmployeeSalaryAtomicRead denotes the action of reading (any) *salary* resource and that Supervisor denotes the role *supervisor*. Then, for any instance of the SecureUML+ComponentUML metamodel that correctly models the access control policy in our running example, $DaAu$(EmployeeSalary-AtomicRead) should return Set{Supervisor}. This is because only supervisors can read employees' salaries.

3 GUI models

In GUI design, it is useful to distinguish between a GUI's visual elements and the behavioural properties associated with these elements. Visual elements are typically called widgets, which are of different types and support different events. Examples of widgets include buttons, entries, containers, and windows. Buttons can be clicked on. Entries can be filled in with text. Containers can graphically contain (group together) other widgets. Windows are a concrete class of containers. The behavioural properties of the different widgets are defined by the actions associated to the events that they support. We distinguish between data actions and widget actions. Data actions act on application data. Widget actions act upon widgets (including themselves).

In this section, we define GUI models as they will be considered throughout this paper. We first provide an abstract definition and afterwards we introduce a specific language for modeling GUIs.

Definition 3. *Let $\mathcal{C}$ be a component-based model $\mathcal{C} = \langle E, At, As, Md\rangle$. Then, a* GUI model *$\mathcal{G}_{\mathcal{C}}$ for $\mathcal{C}$ is a 9-tuple*[7]

$$\mathcal{G}_{\mathcal{C}} = \langle Wd, Wd^{\mathrm{c}}, In, Ev, DaAc, WdAc, WdEv, EvWdAc, EvDaAc\rangle,$$

with $Wd^{\mathrm{c}} \subseteq Wd$, $In : Wd^{\mathrm{c}} \longrightarrow \mathcal{P}(Wd)$, $WdEv : Wd \longrightarrow \mathcal{P}(Ev)$, $EvWdAc : Ev \longrightarrow \mathcal{P}(WdAc)$, and $EvDaAc : Ev \longrightarrow \mathcal{P}(DaAc)$. The model $\mathcal{G}_{\mathcal{C}}$ specifies a

[6] The auxiliary operations compactionPlus() and allRoles() return, respectively, the collection of composite actions to which an action is (directly or indirectly) subordinated and the collection of roles that are (directly or indirectly) assigned to a given permission. We again refer to [1] for the full definitions of these operations.

[7] For the sake of simplicity, we have left implicit the intended dependency of $\mathcal{G}_{\mathcal{C}}$ with respect to $\mathcal{C}$: namely, that the data actions $DaAc$ are indeed actions upon the entities (and their properties) that are modeled in $\mathcal{C}$. To make this dependency explicit, similarly to what we did for the case of security-design models, we would extend our 9-tuple in Definition 3 with two additional components: namely Rs and $RsDaAc$, with $Rs = (E \cup At \cup As \cup Md)$ and $RsDaAc : Rs \longrightarrow \mathcal{P}(DaAc)$.

graphical interface to interact with the component-based system modeled by $\mathcal{C}$*. More concretely,* $\mathcal{G}_{\mathcal{C}}$ *specifies*

- *the widgets* Wd *and widget containers* Wd^{c} *that make up the GUI;*
- *the widgets* $In(wd)$ *that are contained by a given widget container* wd*;*
- *the events* Ev *supported by the GUI;*
- *the data actions* $DaAc$ *and the widget actions* $WdAc$ *that can be triggered by the events;*
- *the events* $WdEv(wd) \subseteq Ev$ *supported by a given widget* $wd \in Wd$*; and,*
- *the widget actions* $EvWdAc(ev) \subseteq WdAc$ *and data actions* $EvDaAc(ev) \subseteq DaAc$ *associated with a given event* $ev \in Ev$*.*

Example 4. The GUI model (partially) specifying the graphical user interface in our running example will, for example, define that: the widgets are four windows, which contain six buttons, one combo-box, three labels and one entry; the buttons support, among others, *on click* events; and the *on click* event supported by the View-button triggers the widget action of opening the Window #3, while the *on click* event supported by the OK-button triggers the data action of *updating* a given *phone number* resource.

The GUI language. This is a simple language for modeling GUIs. The GUI metamodel is shown in Figure 4 (inner rectangle).[8]

Application GUIs consist of widgets that are displayed inside containers, which are themselves widgets. Each widget has a (possibly empty) set of events associated to it, and each event is in turn associated with a set of actions, which are the actions triggered by the event. Also, the events' actions are of two types: widget actions (which are actions on GUI widgets) and model actions (also denoted data actions), which are actions on the application data.

Each valid instance of the GUI metamodel specifies a GUI model $\mathcal{G}_{\mathcal{C}} = \langle Wd, Wd^{c}, In, Ev, DaAc, WdAc, WdEv, EvWdAc, EvDaAc\rangle$, whose components are defined by the following OCL expressions:

$Wd = \mathsf{Widget.allInstances()}.$
$Wd^{c} = \mathsf{Container.allInstances()}.$
$In(wd) = wd.\mathsf{contained}.$
$Ev = \mathsf{Event.allInstances()}.$
$DaAc = \mathsf{DataAction.allInstances()}.$
$WdAc = \mathsf{WidgetAction.allInstances()}.$
$WdEv(wd) = wd.\mathsf{widgetEvents}.$
$EvWdAc(ev) = ev.\mathsf{firedActions{-}{>}select(a|a.oclIsTypeOf(WidgetAction))}.$
$EvDaAc(ev) = ev.\mathsf{firedActions{-}{>}select(a|a.oclIsTypeOf(ModelAction))}$

[8] For the sake of simplicity, our metamodel only defines a basic subsets of widgets (windows, entries, and buttons), of events (entering or leaving a widget, creating a widget, and clicking or double-clicking on a widget), and of widget actions (opening and closing a widget). These subsets are, however, sufficient for the purpose of this paper. The interested reader can find in [7] the definition of a more comprehensive GUI metamodel.

.oclAsType(ModelAction).modelAction).

Example 5. Suppose that onCreateSalaryLabelWindow3 denotes the event that creates a Salary-label in Window #3. Then, for any instance of the GUI metamodel that correctly models the graphical user interface in our running example, $EvDaAc$(onCreateSalaryLabelWindow3) should return Set{EmployeeSalaryAtomicRead}. This is because reading a *salary* resource is the data action triggered when the Salary-label is created.

4 Security-GUI models

We are now ready to provide an abstract definition (Definition 4) of security-GUI models. These are models in which permissions are associated to widget events in order to specify who can execute them. We also introduce a specific language for modeling security-GUI models (SecureUML+GUI).

Definition 4. *Let $\mathcal{C}$ be a component-based model $\mathcal{C} = \langle E, At, As, Md\rangle$. Let $\mathcal{G}$ be a GUI model for $\mathcal{C}$*

$$\mathcal{G}_{\mathcal{C}} = \langle Wd, Wd^{\mathrm{c}}, In, Ev, DaAc, WdAc, WdEv, EvWdAc, EvDaAc\rangle.$$

Then, a security-GUI model $\mathcal{M}_{\mathcal{G}_{\mathcal{C}}}$ for $\mathcal{G}_{\mathcal{C}}$ is a triple $\mathcal{M}_{\mathcal{G}_{\mathcal{C}}} = \langle \mathcal{G}, Rl, EvAu\rangle$, with $EvAu : Ev \longrightarrow \mathcal{P}(Rl)$, that specifies a security policy for accessing the resources (namely, the events) in the graphical interface modeled by $\mathcal{G}_{\mathcal{C}}$. More concretely, it specifies all the roles Rl that users may adopt when interacting with the GUI, as well as the specific roles $EvAu(ev) \subseteq Rl$ that are authorized to execute a given event $ev \in Ev$.

Example 6. The security-GUI model specifying the expected (security) behaviour of the graphical user interface in our running example will, for example, define that the role *administrator* is authorized to execute the events creating the Name and Phone-labels in Window #3, but not the event creating the Salary-label.

The SecureUML+GUI language. SecureUML+GUI is another dialect of SecureUML. It combines SecureUML with our simple GUI modeling language, providing a convenient language for specifying security-GUI models.

The SecureUML+GUI metamodel, shown in Figure 4, provides the connection between SecureUML and GUI. It specifies the protected resources, namely, events, as well as the available actions on these protected resources, namely, their execution.

Each valid instance of the SecureUML+GUI metamodel specifies a security-GUI model $\mathcal{M}_{\mathcal{G}_{\mathcal{C}}} = \langle \mathcal{G}_{\mathcal{C}}, Rl, EvAu\rangle$, whose components are defined by the following OCL expressions:

$Rl =$ Role.allInstances().
$EvAu(ev) = ev$.allAssignedRoles().

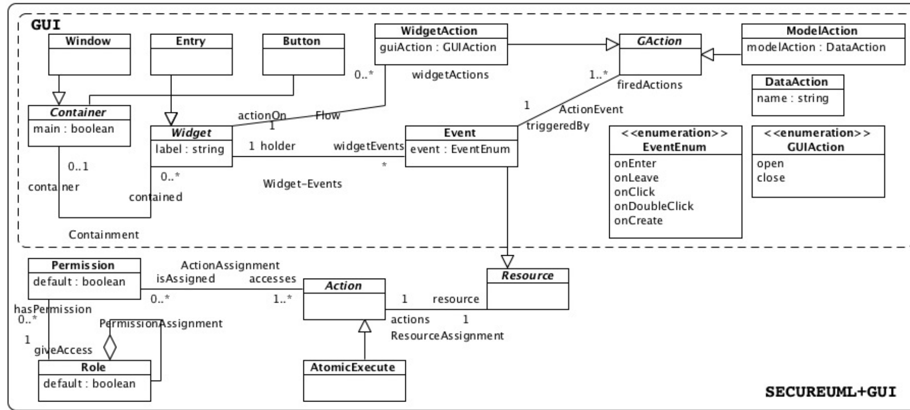

Fig. 4. SecureUML+GUI metamodel.

where the operation allAssignedRoles() is defined as follows:

context Event::allAssignedRoles():Set(Roles)
body: self.actions.isAssigned.allRoles()−>asSet()

Example 7. For any instance of the SecureUML+GUI metamodel that correctly models the graphical user interface in our running example, $EvAu$(onCreateSalaryLabelWindow3) should return Set{Supervisor}. This is because executing the event creating the Salary-label will trigger the data action of reading a *salary* resource, but only supervisors are authorized to execute this data action.

4.1 Smart security-aware GUI models

Informally, a GUI model is smart and security-aware when roles are authorized to execute events *depending on* the actions (both data actions and widget actions) that these events trigger. This dependency relationship, and therefore, the corresponding notion of smartness, can be defined in several ways and we propose one of such definition in this section. Note that our aim is not to give the one "canonical" definition of smartness (should such a definition even exist), but rather to show that non-trivial kinds of smart and security-aware GUI models can be automatically generated from GUI and security-design models using a well-defined model transformation.

To simplify our definition of smartness, in what follows we assume that any model $\mathcal{G}_{\mathcal{C}}$, with

$$\mathcal{G}_{\mathcal{C}} = \langle Wd, Wd^{\mathrm{c}}, In, Ev, DaAc, WdAc, WdEv, EvWdAc, EvDaAc \rangle,$$

satisfies the following properties:

1. There are only two widget actions, namely, opening and closing a widget.

2. Every widget has a distinguished event, namely, the event of creating the widget itself. This event has the following properties:
   – If a widget is a non-container widget, then the only widget action associated to this event is the action of opening the widget itself.
   – If a widget is a container widget, then the only actions associated to this event are the actions of opening the widget itself as well as all the widgets that it (immediately) contains.
3. There are no cycles in the widget opening actions. That is, no widget wd can be opened by an action triggered by an event of a widget wd' that was opened by an action triggered by a sequence of events starting from an event of wd.

We denote by open_{wd} (respectively, close_{wd}) the action of opening (respectively, closing) the widget wd. Also, we denote by $EvWdAc^{\mathrm{o}}(ev)$ the set of opening actions triggered by the event ev, i.e., $EvWdAc^{\mathrm{o}}(ev) = \{\mathrm{open}_{wd} \mid \mathrm{open}_{wd} \in EvWdAc(ev)\}$. Finally, we denote by $\mathrm{onCreate}_{wd}$ the event of creating the widget wd.

Definition 5. *Let $\mathcal{C}$ be a component-based model, $\mathcal{C} = \langle E, At, As, Md\rangle$. Let $\mathcal{S}_{\mathcal{C}}$ be a security-design model for $\mathcal{C}$,*

$$\mathcal{S}_{\mathcal{C}} = \langle Rs, DaAc, Rl, RsAc, DaAu\rangle,$$

with $Rs = (E \cup At \cup As \cup Md)$. Also, let $\mathcal{G}_{\mathcal{C}}$ be a GUI-model for $\mathcal{C}$,

$$\mathcal{G}_{\mathcal{C}} = \langle Wd, Wd^{\mathrm{c}}, In, Ev, DaAc, WdAc, WdEv, EvWdAc, EvDaAc\rangle.$$

Note that DaAc is shared by $\mathcal{S}_{\mathcal{C}}$ and $\mathcal{G}_{\mathcal{C}}$, i.e., the data actions whose access policy is defined by $\mathcal{S}_{\mathcal{C}}$ are exactly those that can be triggered by events in $\mathcal{G}_{\mathcal{C}}$.

Now, let $\mathcal{M}_{\mathcal{G}_{\mathcal{C}}}$ be a security-GUI model for $\mathcal{G}_{\mathcal{C}}$,

$$\mathcal{M}_{\mathcal{G}_{\mathcal{C}}} = \langle \mathcal{G}_{\mathcal{C}}, Rl, EvAu\rangle.$$

$\mathcal{M}_{\mathcal{G}_{\mathcal{C}}}$ *is a* smart and security-aware GUI model with respect to $\mathcal{S}_{\mathcal{C}}$ *if and only if:*

- *The roles that are authorized by EvAu to execute an event ev (different from creating a widget) are exactly those that:*
  - *are also authorized by DaAu to execute all the data actions that will be triggered when executing the event ev, and*
  - *are also authorized by EvAu to create all the widgets that will be opened when executing the event ev (closing a widget, however, is not relevant authorization-wise).*
- *The roles that are authorized by EvAu to create a non-container widget wd are exactly those that are also authorized by DaAu to execute all the data actions that will be triggered when executing this event.*
- *The roles that are authorized by EvAu to create a container widget wd are exactly those that are also authorized by EvAu to create at least one of the widgets (immediately) contained by the widget wd.*

More formally, for any widget $wd \in Wd$ and event $ev \in WdEv(wd)$, the following holds:

- *Case 1: $ev \neq onCreate_{wd}$.*
  Then, $EvAu(ev) =$

$$
= \begin{cases}
\bigcap_{i=1}^{n} DaAu(daac_i) & \text{if } EvDaAc(ev) = \{daac_1, \ldots, daac_n\} \\
 & \text{and } EvWdAc^{\text{o}}(ev) = \emptyset. \\
\bigcap_{i=1}^{n} EvAu(\text{onCreate}_{wd_i}) & \text{if } EvDaAc(ev) = \emptyset \text{ and} \\
 & EvWdAc^{\text{o}}(ev) = \{\text{open}_{wd_1}, \ldots, \text{open}_{wd_n}\}. \\
(\bigcap_{i=1}^{n} DaAu(daac_i)) \bigcap & \\
(\bigcap_{i=1}^{m} EvAu(\text{onCreate}_{wd_i})) & \text{if } EvDaAc(ev) = \{daac_1, \ldots, daac_n\} \text{ and} \\
 & EvWdAc^{\text{o}}(ev) = \{\text{open}_{wd_1}, \ldots, \text{open}_{wd_m}\}.
\end{cases}
$$

- *Case 2: $ev = onCreate_{wd}$.*
  Then, $EvAu(ev) =$

$$
= \begin{cases}
Rl & \text{if } wd \notin Wd^{\text{c}} \\
 & \text{and } EvDaAc(ev) = \emptyset \\
\bigcap_{i=1}^{n} DaAu(daac_i) & \text{if } wd \notin Wd^{\text{c}} \\
 & \text{and } EvDaAc(ev) = \{daac_1, \ldots, daac_n\}. \\
\bigcup_{i=1}^{n} EvAu(\text{onCreate}_{wd_i}) & \text{if } wd \in Wd^{\text{c}} \\
 & \text{and } In(wd) = \{wd_1, \ldots, wd_n\}.
\end{cases}
$$

Lemma 1. *Let $\mathcal{C}$ be a component-based model, $\mathcal{C} = \langle E, At, As, Md \rangle$. Let $\mathcal{S}_{\mathcal{C}}$ be a security-design model for $\mathcal{C}$,*

$$\mathcal{S}_{\mathcal{C}} = \langle Rs, DaAc, Rl, RsAc, DaAu \rangle,$$

with $Rs = (E \cup At \cup As \cup Md)$. Also, let $\mathcal{G}_{\mathcal{C}}$ be a GUI-model for $\mathcal{C}$,

$$\mathcal{G}_{\mathcal{C}} = \langle Wd, Wd^{\text{c}}, In, Ev, DaAc, WdAc, WdEv, EvWdAc, EvDaAc \rangle.$$

Then, there exists a unique security-GUI model for $\mathcal{G}_{\mathcal{C}}$ that is smart and security-aware with respect to $\mathcal{S}_{\mathcal{C}}$. This model is denoted by $\mathcal{M}_{(\mathcal{G}_{\mathcal{C}}, \mathcal{S}_{\mathcal{C}})}$,

Proof. Due to space limitations, we only sketch the proof of this lemma here. The crucial point is that, given a security-GUI model $\mathcal{M}_{\mathcal{G}_{\mathcal{C}}} = \langle \mathcal{G}_{\mathcal{C}}, Rl, EvAu \rangle$, the clauses in Definition 5 precisely define, for any event supported by $\mathcal{G}_{\mathcal{C}}$, the set of roles in Rl that should be returned by $EvAu$, if $\mathcal{M}_{\mathcal{G}_{\mathcal{C}}}$ is to be considered smart and security-aware with respect to $\mathcal{S}_{\mathcal{C}}$. First, the clauses in Definition 5 cover all the possible cases. In fact, for every event ev, there is a clause (and only one) that applies to this event. Then, every clause either identifies a specific set of roles as the expected result for $EvAu$ or it recursively calls $EvAu$ on some creating events (namely, those associated to the widgets that will be open when executing the event). Since our GUI models are finite and neither include cycles in the widget opening action nor in the containment-relationship, these recursive calls always terminate. Consequently, for any $\mathcal{S}_{\mathcal{C}}$ and $\mathcal{G}_{\mathcal{C}}$, there always exists a security-GUI model for $\mathcal{G}_{\mathcal{C}}$, namely $\langle \mathcal{G}_{\mathcal{C}}, Rl, EvAuSmart \rangle$, that is smart and security-aware with respect to $\mathcal{S}_{\mathcal{C}}$, where $EvAuSmart : Ev \longrightarrow \mathcal{P}(Rl)$ is the function defined by the clauses in Definition 5 for the given $\mathcal{S}_{\mathcal{C}}$ and $\mathcal{G}_{\mathcal{C}}$.

5 Automatically generating smart, security-GUI models

In this section, we sketch the definition of a QVT operational transformation `smartandsecure()` that, given a SecureUML+ComponentUML model and a GUI model, automatically generates a SecureUML+GUI model that is both smart and security-aware. The crucial step in this transformation is, of course, the creation of the Permission-objects, and how they link Role-objects to Atomic-Execute-objects (and, through them, to Event-objects). Recall that for the generated model to be considered smart and security-aware, for any of its Event-objects, the value returned by the operation $\mathsf{allAssignedRoles()}{-}{>}\mathsf{asSet()}$, which "navigates" through those links, must satisfy Definition 5.

We split `smartandsecure()` into two sequential auxiliary model transformations: `generatemodel()` and `addpermissions()`. The method `generatemodel()` generates a SecureUML+GUI model $\mathcal{M}$ that, basically, contains all the widgets, events, widget actions, and data actions, along with their links, that are specified in a given (source) GUI model $\mathcal{G}_{\mathcal{C}}$, plus the roles Rl that are specified in a given (source) SecureUML+ComponentUML $\mathcal{S}_{\mathcal{C}}$. More formally, `generatemodel()` generates a SecureUML+GUI model $\mathcal{M}_{\mathcal{G}_{\mathcal{C}}} = \langle \mathcal{G}_{\mathcal{C}}, Rl, EmptyEvAu\rangle$, where $EmptyEvAu(ev)$ returns, for any event considered in $\mathcal{G}_{\mathcal{C}}$, the empty set of roles, i.e., the expression $ev.\mathsf{allAssignedRoles()}{-}{>}\mathsf{asSet()}$ evaluates to $\mathsf{Set}\{\}$. The generated model $\mathcal{M}_{\mathcal{C}}$ is not yet security-aware or smart (unless the given security-design model $\mathcal{S}_{\mathcal{C}}$ does not specify any authorization restrictions.)

Next, based on the results of evaluating an OCL operation `EvAuSmart()` (whose definition directly translates into OCL the clauses in Definition 5), the method `addpermission()` augments the SecureUML+GUI model $\mathcal{M}_{\mathcal{G}_{\mathcal{C}}}$ with all the permissions that makes this model smart and security-aware with respect to $\mathcal{S}_{\mathcal{C}}$. More concretely, it adds all the permissions, along with their links to the appropriate roles and atomic execute actions, that are required for the following to hold: for any event ev, the expression $ev.\mathsf{actions.isAssigned.allRoles()}{-}{>}\mathsf{asSet()}$ evaluates to the set of roles that should be authorized to execute the event ev if $\mathcal{M}_{\mathcal{G}_{\mathcal{C}}}$ is to be considered smart and security-aware with respect to $\mathcal{S}_{\mathcal{C}}$.

6 Related and Future Work

Creating user interfaces is a common task in application development and one that is often time consuming and therefore expensive. 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 until now on the systematic design of GUIs whose functionality should adhere to the security policy of the underlying application-data model. The idea we develop here originated in [12], where we first proposed using model-transformations to generate simple (not necessarily smart) security-aware GUI models.

In the modeling community, other researchers have investigated how to extend existing modeling languages for GUI modeling. [4] proposes a UML profile to model GUI layout. We do not consider layout issues since translating security from data models into GUI models is independent (except for containment

hierarchies, which we do consider) of the graphical appearance and location of the widgets. However, we do plan to use widgets' graphical information to make our GUIs more appealing to users. [3] proposes a heavyweight, template-based extension of UML for GUI modeling to help develop GUIs for large-scale systems, although access control decisions are not part of this work either. This work is similar to ours in that it also separates concerns within the modeling phase by separating the construction of the application and the GUIs. Our modeling techniques also differ since we use metamodeling in contrast to their use of stereotypes. In [3], the task of the GUI designer is reduced to choosing between *window types*, which are parameterized templates (including interaction behavior and layout). If we had used stereotyped GUI designs in our approach, we would have provided different fixed GUI designs that could be made automatically smart and security-aware according to the underlying data model security policy.

[6] approaches security as a *crosscutting concern* in terms of aspect-oriented programming for a software system. The main contribution of this work is a behavioral definition of aspects. The authors propose enriching a data model with a security policy by performing a model transformation using the bidirectional object-oriented transformation language (BOTL) [9, 10]. Although they present examples, they only outline a method for model transformation based on the proposed definition. Therefore a comparison here would be difficult. Notice also that such comparison would concern our integration of the SecureUML policies in the data model, not the GUI model generation itself. In [13], the authors present a model transformation methodology to integrate non-functional requirements, such as security, in a model-driven software product line. In this setting, abstract design models of the application and its security are built and these models are refined in parallel using model transformation to obtain an implementation model with Java Platform, Enterprise Edition (JEE) security annotations. In contrast, we integrate the data, the GUI, and the security platform-independent models (PIMs) from the design stage to obtain a security-aware GUI PIM from which one can generate a functional GUI, which will provide a security-aware and smart access to the data. Regarding GUI generation, there are a number of tools for the automated design and generation of user interfaces, which can generate the static layout of an interface from the application's data model. However, security concerns are not among the problems addressed.

In the programming community, independent of model-driven initiatives, numerous projects have addressed the problem of how to best implement graphical user interfaces for application data. For example, [8] proposes enriching the application source code with annotations that control the generation of the graphical user interfaces. Other researchers have designed and implemented specialized tools that generate 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 into IDEs or available as plug-ins, that simplify the task of creating application GUIs in different programming languages. However, to the

best of our knowledge, [7] is the only tool capable (although still a prototype) of automatically generating smart, security-aware functional GUIs.

Finally, our work is also related to research in the field of intelligent user interfaces. In our view, smart security-aware GUIs can be seen as a class of intelligent user interfaces. Our GUIs take advantage of the users' status to tailor their access to the application data. An interesting follow up question concerns the generality of our model-transformation approach and whether it can be used to generate other classes of intelligent interfaces.

7 Conclusions

We have presented an approach based on model-transformation for automatically generating smart, security-aware GUIs. Given an application-data model and a GUI model, our transformation makes the GUI model both smart and security-aware. We have implemented our approach using the Operational QVT transformation engine that is provided within Eclipse.

As a design methodology, our approach has three main advantages over traditional approaches to software design. 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 generated 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 work presented here is the corner stone of a more ambitious project for making model-driven security an effective and useful approach for generating multiple layers of security-critical systems in industrial software development.

References

1. D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. *Information and Software Technology*, 51(5):815–831, 2009.
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. 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.
5. 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.
6. J. Fox and J. Jürjens. Introducing security aspects with model transformations. In *12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), 4-7 April 2005, Greenbelt, MD, USA*, pages 543–549, 2005.

7. BM1 Software Group. The SmartGUI Project. `http://www.bm1software.com/`, 2009.
8. 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.
9. F. Marschall and P. Braun. Model transformations for the MDA with BOTL. Technical report, University of Twente, 2003.
10. F. Marschall and P. Braun. Bidirectional object oriented transformation language. `http://sourceforge.net/projects/botl/`, 2005.
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. M. Schläpfer, M. Egea, D. Basin, and M. Clavel. Automatic generation of security-aware GUI models. In Alessandra Bagnato, editor, *European Workshop on Security in Model Driven Arquitecture 2009 (SEC-MDA 2009)*, number WP09-06 in Workshop Proceedings Series, pages 42–56. CTIT, Enschede, the Netherlands, 2009.
13. A. Yie, R. Casallas, D. Deridder, and R. Van Der Straeten. Multi-step concern refinement. In *EA '08: Proceedings of the 2008 AOSD workshop on Early aspects*, pages 1–8, New York, NY, USA, 2008. ACM.