

Automatic Generation of Security-Aware GUI Models

Michael Schläpfer Marina Egea David Basin
Manuel Clavel

ETH Zürich, Switzerland

IMDEA Software Institute, Madrid, Spain

Universidad Complutense de Madrid, Spain

24 June 2009

Outline

- 1 Motivation, Proposal, Process and Advantages
- 2 The Transformation Model Types
- 3 The Security Transformation
- 4 Analysis
- 5 Related Work
- 6 Future Work and Conclusions

Motivation: A past industrial experience

Goal: to enhance the test report configuration (TRC) facility provided by an automatic test system they commercialized.

Methodology: we produced the security-design model of the TRC utility capturing the access control requirements.

Problem: manually lift the access control policy from (the code implementing) the persistent layer to (the code implementing) the GUI layer.

[M. Clavel, V. Silva, C. Braga, and M. Egea. Model-driven security in practice: An industrial experience. In I.

Schieferdecker and A. Hartman, editors, *ECMDA-FA 2008: Proceedings of Model Driven Architecture - Industrial Track*, volume 5095 of *LNCS*, pages 327–338, Berlin—Heidelberg, 2008. Springer—Verlag.]

Problem and Proposal

The problem: There is an important, but little explored, link between visualization and security:

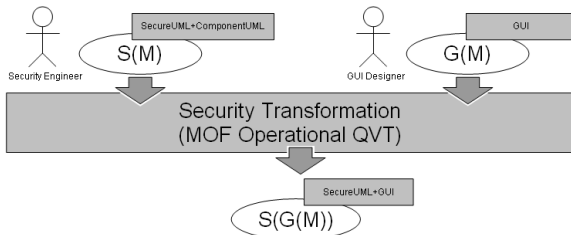
- when the application data is protected by an access-control policy, the application GUI should be *aware of* and *respect* this policy.

E.g., a widget should not give users options to execute actions on the application data that they are not authorized to execute.

Our proposal: We define a many-models-to-model transformation that, given a security-aware data model and a GUI model, makes *automatically* the GUI model also security-aware.

Design process:

- 1 Software engineers specify the application data model M .
- 2 Security engineers specify, in the security model $S(M)$, the application-data access-control policy,
- 3 GUI designers specify the application GUI model $G(M)$.



- 4 Finally, the application security-aware GUI model $S(G(M))$ is automatically generated

Separation of Concerns:

Our model-transformation based approach has three principle advantages over traditional software development approaches.

- 1 Security engineers and GUI designers can *independently* model what they know best.
- 2 Security engineers and GUI designers can independently change their models. 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 check its impact on their designs. They can check the security-aware GUI models to know whether they are designing the right GUI to give the (authorized) users access to the (intended) data.

What does model transformation mean?

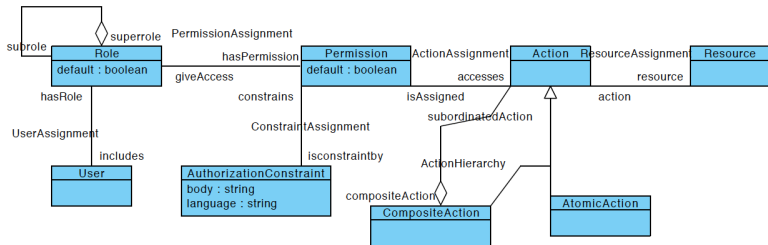
Model transformation is the process of converting some models M_1, \dots, M_n (the transformation *source* models) into other models M'_1, \dots, M'_m (the transformation *target* models).

Transformation model types are provided by metamodels.

- The *types* of our source models are SecureUML+ComponentUML and GUI models and the *type* of our target models is SecureUML+GUI.

SecureUML

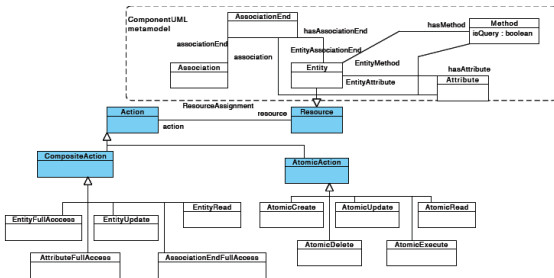
SecureUML is a language based on RBAC for modeling access-control policies on protected resources.



Protected resources are declared as subclasses of *Resources* and the actions that they offer to clients as subclasses of *Atomic* or *Composite Actions*.

SecureUML+ComponentUML

This dialect combines SecureUML with a simple language for modeling component-based systems.

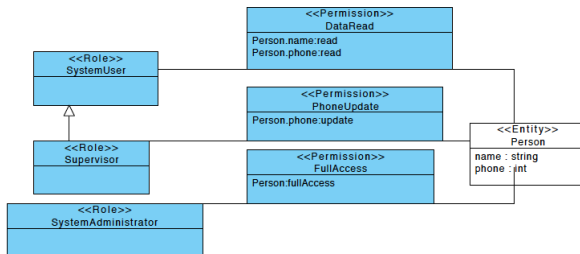


The dialect connection specifies the protected resources and the actions on these protected resources and their hierarchies.

Example: The PhoneBook application

In this application, each entry in the phone directory consists of a name and a phone number. The access policy is:

- Users can read people's name and phone numbers.
- Supervisors are users who can also change phone numbers.
- Administrators can read, write, create and delete entries.



A metamodel based approach to analyze models

- In general, given a modeling language with a formal semantics, one can reason about their models by reasoning about their semantics.
- Our metamodel-based approach reduces *deduction* to the *evaluation* of OCL queries on the corresponding metamodel.

[D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. In *Information and Software Technology Journal*, Elsevier. Special issue on *Model Based Development for Secure Information Systems*, 2008.]

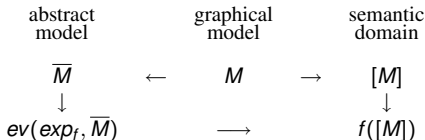
A metamodel based approach to analyze models

Let \mathcal{M} be a modeling language.

- we formalize the desired properties as OCL queries using elements provided by the metamodel of \mathcal{M} ;
- we evaluate these queries on the corresponding snapshots \overline{M} of the metamodel of \mathcal{M} .

A metamodel based approach to analyze models

- we require a *mapping* from graphical models M to abstract models \overline{M} , and that the OCL evaluation, correctly interacts with the interpretation function $[\cdot]$ in the following sense:



$[\cdot]$ is the interpretation function.

f is a function on the semantic domain.

exp_f is an expression to formalize f in OCL.

Analysis with OCL

- Given a role, what are the roles (directly or indirectly) *above* it in the role hierarchy

```
context Role::superrolePlus():Set(Role) body:  
self.superrolePlusOnSet(self.superrole)
```

```
context Role::superrolePlusOnSet(rs:Set(Role))  
:Set(Role) body:  
  if rs.superrole->exists(r|rs->excludes(r))  
  then self.superrolePlusOnSet(rs  
    ->union(rs.superrole)->asSet())  
  else rs->including(self)  
  endif
```

Analysis with OCL

- Given a role, what are the directly or indirectly permissions assigned to it?

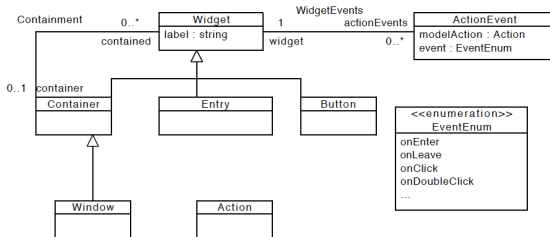
```
context Role::allPermissions():Set(Permission) body:  
self.superrolePlus().haspermission->asSet()
```

- Given a role, what are the atomic actions that a user in this role can perform?

```
context Role::allAtomicActions():Set(Action) body:  
self.allPermissions().allAction()->asSet()  
->select(a|a.ocIsKindOf(AtomicAction))
```

GUI metamodel

It is a simple language for modeling GUIs. We assume that



- each event can trigger one action on the application data;
- the *modelAction* attribute value is an atomic action on a resource of the SecureUML+ComponentUML application-data model.

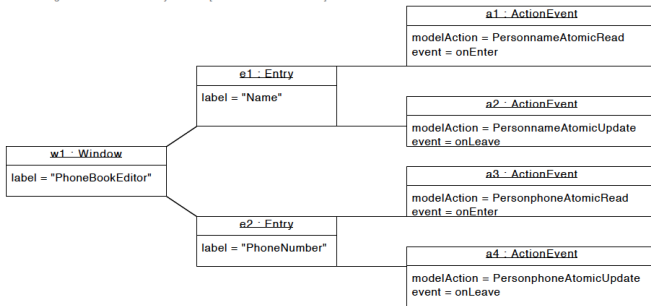
Example: The PhoneBook Editor GUI

The GUI designer designs a GUI consisting of a window *PhoneBook Editor*, with two entry boxes: *Name* and *Phone Number*. At the time of creation, each instance of *PhoneBook Editor* is associated to an instance P of *Person*. Moreover:

- On entering the entry boxes *Name* and *PhoneNumber*, they should display resp. the name and the phone number of the object P .
- On leaving the entry boxes *Name* and *PhoneNumber*, the text displayed in them should be used to update the attributes *Name* and *PhoneNumber* of the person P .

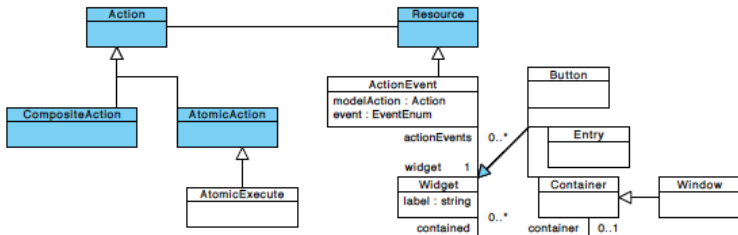
Example: The PhoneBook Editor Model (AS)

Next figure shows the instance of the GUI metamodel that corresponds to the GUI model specifying the previous design.



SecureUML+GUI

This language combines SecureUML with GUI.
SecureUML+GUI models specify who can execute which events on which widgets.



ActionEvents are the protected resources and *AtomicExecute* actions are the possible actions on them.

The idea

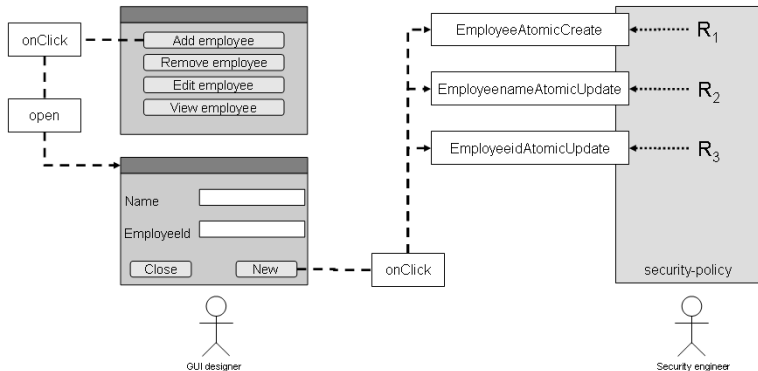


Figure: An example GUI before the Security Transformation.

The idea

For every role that is allowed to perform every model action assigned to an event, a permission is generated granting the role access on the *AtomicExecute* action that is assigned to this event.

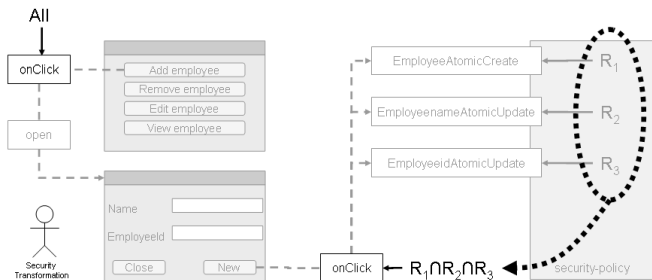


Figure: An example GUI during the Security Transformation.

Description of the Security Transformation (I)

Step 1: The model elements of the target model are created:

- The *Roles* in the (source) SecureUML+ComponentUML model are copied, along with their hierarchies, in the (target) SecureUML+GUI model.
- The *Widgets* in the (source) GUI model are copied, along with their containment relationships and their associated *ActionEvents*, in the (target) SecureUML+GUI model.
- 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.

```
\*Step 1*\ transformation sectrans(in gui : GUI,  
    in secomp : SEC+COMP, out secgui : SEC+GUI);  
main() {  
    secomp.objects() [SEC+COMP::Role]->map Role_to_Role();  
    secomp.objects() [SEC+COMP::Role]->map  
        preserve_Role_hierarchy();  
    gui.objects() [GUI::ActionEvent]->map  
        ActionEvent_to_ActionEvent();  
    gui.objects() [GUI::Window]->map Widget_to_Widget();  
    gui.objects() [GUI::Entry]->map Widget_to_Widget();  
    gui.objects() [GUI::Button]->map Widget_to_Widget();  
    gui.objects() [GUI::Widget]->map  
        preserve_containment_hierarchy();  
    secgui.objects() [SEC+GUI::ActionEvent]->map  
        addAtomicExecuteAction();  
}
```

Description of the Security Transformation (II)

Step 2: The permission assignments in the target model are created:

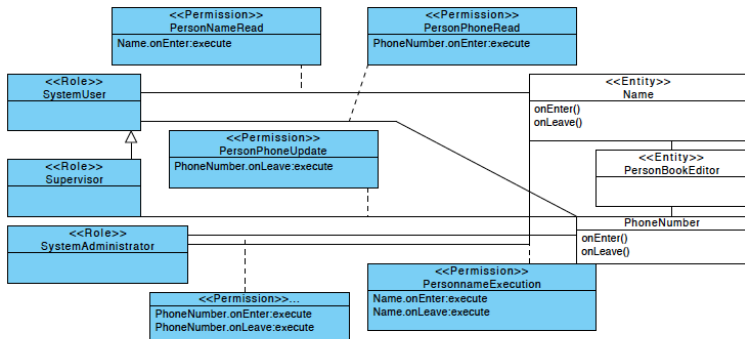
- For each *ActionEvent*'s *action* in the (source) GUI model, and for each *Role* that is allowed to perform this *action* in the (source) SecureUML+ComponentUML model, a *Permission* is created in the (target) SecureUML+GUI model that grants access to *Role* to execute the *ActionEvent*'s *event*.

```
/* Step 2: */  
guiSecPolicy.objects() [SECUMLANDGUI::ActionEvent]  
->liftPermissions();
```

The operation *liftPermissions()* uses in its definition the operation *allAtomicActions()*.

Example: The output security aware GUI model

In the figure, we show the security aware GUI model that results from applying our many-models-to-model transformation.



Analysis of the transformation

Let rl be a *Role* in $S(M)$. Let $acev$ be an *ActionEvent* in $G(M)$.

- **Correctness analysis:** 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.

$rl.allAtomicActions() \rightarrow includes(acev.action)$ evaluates to **true** in $S(G(M))$ only if

$rl.allAtomicActions() \rightarrow includes(acev.modelAction)$ evaluates to **true** in $S(M)$.

Analysis of the transformation

Let rl be a *Role* in $S(M)$. Let ac be an *AtomicAction* in $S(M)$.

- **Completeness analysis:** we can check whether for every atomic action that a role is allowed to execute in the $S(M)$ model, it exists an event associated to this action by an *ActionEvent* object whose corresponding atomic execute action can be performed by that role in the $S(G(M))$ model. $rl.allAtomicActions() \rightarrow includes(ac)$ evaluates to **true** in $S(M)$ only if

```
ActionEvent.allInstances() -> select(acev |  
acev.modelAction=ac.name) ->  
collect(acev | acev.action) ->  
exists(a | rl.allAtomicActions() -> includes(a)) evaluates to  
true in  $S(G(M))$ 
```

Related Work

- In the MDA community:
 - proposals of UML extensions for GUI modeling;
 - surveys reviewing the tools supporting general modeling, model transformations, model constraints, etc., in relation to the needs of HCI;
 - a part of the Sectet-framework supports a DSL for the design of inter-organizational workflows along with several security patterns. They transform PIM to PS artefacts of a web-services based architecture using QVT. They exemplify how model-to-code transformation could be implemented with openArchitectureWare.

Related Work

- In the programming community, there are several projects to implement GUIs for application data:
 - enriching the application source source with annotations to control the generation of GUIs;
 - tools to support the automatic generation of GUIs meeting specific requirements;
 - there are GUI builders within IDEs or available as plug-ins, to ease the creation of GUIs in different programming languages.

Future Work

- 1 Lift up security policies that also depend on the satisfaction of authorization constraints in a system state.
- 2 Work with GUI models that associate multiple actions to single events.
- 3 Generate *smart*, security-aware GUI models by a twofold permission “lifting”:
 - i from widgets whose events triggered actions on application data to widgets whose events triggered actions on those widgets and,
 - ii from widgets that are contained to the widgets that contain them.
- 4 Generate the actual GUIs from our final models following the MDA philosophy.

Ongoing and Completed work

- Previous items 1, 2 and 3 are already done, have been submitted for publication and are part of a master thesis delivered two weeks ago to ETHZ that is currently under evaluation.
- Previous item 4 is ongoing work: we already have encouraging results generating security aware Java GUIs automatically from our output models using the QVTO and JET technologies of the Eclipse platform.

Conclusions

- This work is the corner stone of a more ambitious project for making MD security an effective and useful approach for generating multiple system layers as part of a security-intensive industrial software development.
- We directly pursue *conformance*, meaning that executing events on the GUI layer never leads to program exceptions from the access-control security policy implemented at the persistent layer.

Conclusions

We aim to provide GUI designers with better models and tools

- for building and analyzing GUIs for security-critical applications,
- for automatically checking that, using a given GUI, (authorized) users can indeed access the (intended) application data,
- 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.