ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Masters Thesis

Automatic Generation of Smart&SecGUIs from Security Design Models

by
Michael Schläpfer

Due date
10. June 2009

Advisors:
Dr. Marina Egea
Prof. Dr. David Basin

ETH Zurich, Information Security Group
Department of Computer Science
8092 Zurich, Switzerland

Abstract

Model-Driven Architecture (MDA) holds the promise of reducing system development time while improving the quality of the resulting products. It is argued that the construction of models during requirements analysis and system design will improve the quality of the resulting systems by providing a foundation for early analysis and fault detection. Moreover, the models constructed in the analysis and design phases will serve as specifications for the later development phases and, when they are sufficiently formal, they will also provide the basis for refinement down to code through well-defined model transformation functions. Model-Driven Security (MDS) is a recently proposed specialization of the MDA approach. Here, "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." It is argued that this approach "bridges the gap between security analysis and the integration of access control mechanisms into end systems. Moreover, it integrates security models with system design models and thus yields a new kind of model, *security design models*." Recent investigations have shown that security policies can be integrated into system design models and that the resulting security design models can be used as a basis for generating systems along with their security infrastructures. Indeed, reports of successful implementations of security design models in real industrial projects and several illustrative modeling examples can be found in literature. This thesis presents a contribution to this research field.

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* this policy. For example, the GUI should not display options to users for actions that they are not authorized to execute on application data. Directly hard-coding the security policy within the GUI is inadequate. GUI designers are not (and usually should not be) aware of the application data security policy. In this thesis we propose a methodology based on model transformations to automatically make a GUI model designed for a protected data model aware of its security policy. We define many-models-to-model transformations that, given a security-aware data model and a GUI model, make the GUI model security-aware and also smart, security-aware. For example, the resulting GUI widgets will not give users the option to open other widgets when this would allow users to execute unauthorized actions on the application data. Overall, we aim to provide GUI designers with better models and tools for building and analyzing GUIs for security-critical applications.

We implemented the model transformations proposed in this thesis as a proof of concept using concrete implementations of the QVT specification. These concrete implementations, together with some examples, are available in the appendix of this work.

Preface

The high-level idea of a future development process for automatically support developers in their tasks to build security-intensive applications, including multiple system layers, fascinated me and still does. After spending a lot of time into this topic I am thankful to my supervising professor David Basin for giving me the opportunity to learn more about the concepts of model-driven security by suggesting me to do this work.

As the topic of automatically generate smart, security-aware GUIs 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, this thesis gave me the possibility to directly interact with some leading people in this field. Therefore working on this project did not just give me the opportunity to learn a lot about modeling and model transformations, but also about working conceptually and integrated into a research team. The fact that I contributed to two different research papers during this masters thesis was an exciting experience as well and provided me with some valuable insights into the world of science. Indeed, several concepts developed during this master thesis, have been further improved during the work on the above mentioned scientific papers by the continuous interaction between Marina Egea, Manuel Clavel, David Basin and myself.

I want to point out some special thanks to my supervisor Marina Egea who gave me invaluable inputs and feedbacks. The fruitful discussions helped me a lot to improve my considerations and make progress in my work. I also would like to thank Manuel Clavel from the IMDEA Software Institute of Madrid in Spain as well as David Basin, my supervising professor here at ETH. Both of them had an open ear every time I had questions and supported my work by raising interesting questions during our meetings.

Finally, I want to thank my girlfriend Kathrin for her patience during this phase and my and her parents for their support during my whole studies at the Swiss Federal Institute of Technology.

Contents

Chapter 1

Introduction

1.1 Motivation

In many software applications, users access application data using GUI widgets: data are 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 he lacks permissions to execute the actions that are required on the application data. To see how this link between GUIs and security policies might look, consider a simple example: a *smart, security-aware* GUI for managing information about a company's employees. Suppose the main window is a menu with options for adding and removing employees, and for editing and viewing employee information. The window for adding employees will then include entries for entering the data required to create a new employee, e.g., name and employee identification number, a button for closing the window, and a button for actually creating the new employee with the information provided in the entry boxes. Since adding employees is a task typically reserved for members of the Human Resources (HR) department, the main window should not offer this option to members of other departments within the company. Figure 1.1 illustrates the desired behavior for this simple GUI. The left-hand side *a*) shows the window offered to members of the HR department while the right-hand side *b*) shows the window displayed to all other employees. For the sake of simplicity, we are only illustrating "smart, security-awareness" with respect to the security policy governing the addition of employees. Similar behavior would be needed with respect to the policies for removing employees and editing and viewing employee information. To realize GUIs like the above, traditional GUI functionality must be combined with awareness of the security policy. How should this combination be realized? The traditional job of a GUI designer is to focus on the GUI's layout and its behavior, i.e., which events trigger which actions on which application data and application widgets. In our simple example, the GUI designer would draw the layouts for the root window ('Main Menu'), for the windows for adding ('Add

Figure 1.1: An example window. a) view of HR members; b) view of other employees.

Employee') and removing employees, and for editing and viewing employees' information. The GUI designer would also be responsible for connecting the widgets with appropriate events in the application, e.g., annotating each widget with the events (if any) that will trigger actions on the application data or other application widgets, as well as with the specific actions that will be triggered by each of these events. For example, the button 'Add Employee' in Figure 1.1 should have attached a note indicating that after clicking on the button, a new window will open that provides then the possibility to *create* a new employee. We do not propose that the GUI designer should do more than the above and also be responsible for integrating the security policy in the GUI. To begin with, GUI designers are not (and usually should not be) aware of the application data security policy. Moreover, it would be unreasonable to expect the GUI designer annotate, additionally, each widget with the information about who can execute each widget's events. To do this, the GUI designer would have to take into consideration the security-related annotations attached to all the widgets that the actions associated to the widget's events may lead to. This would be both cumbersome and prone to errors if done manually.
In this thesis, we propose a new approach for designing application GUI models that are smart and security-aware. Rather than having the GUI designer build-in these features by hand, we will automate these additions. Our approach is based on model transformations in a model-driven development setting: we define a many-models-to-model transformation that given a security-aware data model and a GUI model, automatically annotate the later with all the relevant security information. This model transformation is the key component of our proposal for designing smart, security-aware application GUI models. Following an MDA approach, GUI code can then be automatically generated from the resulting models that will, by design, have the expected properties.[1] We depict our methodology in two steps in figure 1.2. In a first step 1.), we propose a

[1]However, we do not cover here the issue of code-generation from smart, security-aware application GUI models. See [6] for a report of an experience on following a (less elaborated) model-driven security approach on an industrial project.

methodology to make GUI models security-aware. Then in the second step 2.), we refine that methodology to generate GUIs that are now both smart and security-aware. In both approaches we assume the following processes for the design of security-aware GUIs and smart, security-aware GUIs respectively:

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, and GUI designers specify the application GUI model $G(M)$.
3. The application smart, security-aware GUI model $S(G(M))$ is automatically generated from the security model $S(M)$ and the GUI model $G(M)$.

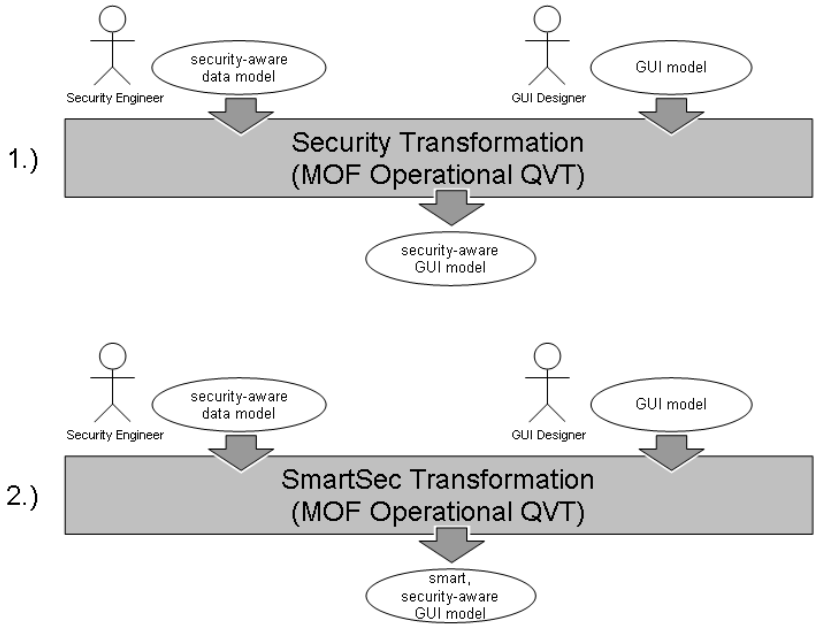

Figure 1.2: Generating smart, security-aware application GUIs.

Our model-transformation based approach for designing smart, security-aware GUI models has the following 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 final smart, security-aware GUI models.
- Finally, 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 smart, 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 smart, 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 address with our model-transformation based approach, is *compliance*. For the case considered in this thesis, compliance means that executing events on the GUI layer never leads to program exceptions from the access control security policy implemented at the persistence layer.

1.2 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, [23, 4, 7] propose various UML extensions for this purpose. There are also approaches [12] 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 [26] to map the data contained in the underlying data model to the model implemented by the widget. More directly related to MDA, [18] 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 [19], which focuses on transformation tools for model-based user interface development. In relation to security, [21, 22] uses QVT to handle security requirements in an MDA setting; in particular, to obtain the secure logical scheme from conceptual models. Also, [11] 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, [13] 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 [17]. 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. Although less immediately related to our present work, there are also tools that ease the task of implementing RBAC policies [10], for instance, the AccesMgr tool (for managing access control lists for Windows NT files), or the RGPAdmin tool (for managing role/permission relationships in the RBACWeb tool).
Finally, our work is also related to research in the intelligent user interface field. In our view, smart, security-aware GUIs can be seen as a class of intelligent user interfaces: they 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 to generating other classes of intelligent interfaces.

1.3 Focus and Contributions of this work

The focus of this Master Thesis was to capture proper concepts of *security-awareness* and *smartness* for GUIs that, given a SecureUML dialect for an application data model and a GUI design for such model, would provide and guide the implementation of a model-to-model transformation that automatically makes the GUI design security-aware and smart. Indeed, we provide a formal definition of what does security-awareness and smartness mean in this setting, their implementations by model transformations and other contributions that we summarize as follows:

- In chapter 3 we define a SecureUML dialect for GUIs, based on our own defined GUI metamodel, which although simple has revealed sufficient to deal with significant case studies. We also provide query operations defined in OCL to automatically analyze such models.

- In chapter 4 we provide a novel formal definition of security-aware GUIs: that is, define the properties that a GUI for an application (a SecureUML+-GUI model) must satisfy in order to be a security-aware GUI model. We also provide functions to compute the access control information for each GUI element according to the underlying security policy protecting the data model. We implement these functions in QVT so we automate the computing process to obtain security aware GUIs. We show that the model-to-model transformation is such that: it only generate security-aware GUIs; it preserves the security-awareness: that is, what was forbidden (actions on the ComponentUML resources) before, is still forbidden; and what was allowed (actions on the ComponentUML resources) is still allowed. Finally, we provide a method to validate the definition of security-aware GUIs by checking the corresponding OCL expressions over the SecureUML+GUI models that capture the security aware GUI definitions.

  The main technical results of this chapter are accepted for its publication in the proceedings of the Security in Model Driven Architecture (SEC-MDA'09) workshop that will be held in Enschede (The Netherlands) in

June 24th, 2009 [20], co-located with the Fifth European Conference on Model Driven Architecture - Foundations and Applications (ECMDA'09).

- In chapter 5 we provide a novel formal definition of smart, security-aware GUIs: that is, define the properties that a GUI for an application (a SecureUML+GUI model) must satisfy in order to be a smart, security-aware GUI model. We also provide functions to compute the access control information for each GUI element according to the underlying security policy protecting the data model and the organizational architecture of the GUI design. We implement these functions in QVT so we automate the computing process to obtain smart, security aware GUIs. We remark that the model-to-model transformation is such that: it only generate smart GUIs while preserving security-awareness. Finally, we provide a method to validate the definition of smart, security-aware GUIs by checking the corresponding OCL expressions over the SecureUML+GUI models that capture the smart, security aware GUI definitions

  The main technical results of this chapter have been submitted for publication [2].

- In chapter 6 we draw conclusions and point out some directions of future work.

1.4 Outline of the Thesis

Chapter 2: Background. This chapter provides an introductory overview of the technologies and concepts used in this work in order to give the reader the needed background information for understanding the methodologies introduced in later chapters. We describe model-driven security in general and in this context how the SecureUML language can be combined with different design languages. Also, we introduce the OCL language that we use later on to analyze models. Finally, we introduce the concepts of model transformations using MOF QVT and a case study that will serve as a guiding example for the remaining chapters.

Chapter 3: Modeling Secure GUIs. In this chapter we first give an overview of GUI modeling and present a typical GUI modeling process. Then we introduce a simple GUI modeling language and how it is combined with SecureUML, i.e., the SecureUML+GUI language that we use as the modeling language to automatically generate security-aware and smart, security-aware GUI models, respectively. Also, we explain how SecureUML+GUI models can be automatically analyzed by the definition and evaluation of OCL operations.

Chapter 4: Security-Awareness. In this chapter we provide a brief description of what can be understood by a security aware GUI design. We present a formalization of GUI designs' artifacts and formally state our definition of security aware GUIs. We construct functions which compute the access-control information needed to annotate GUI's widgets according to the underlying security policy holding on the data model. We show how these functions correctly fulfill our definition and help us to specify our QVT security transformation which is also described in this chapter.

Chapter 5: Smartness. In this chapter we provide a brief description of what can be understood by a smart, security aware GUI design. We formally state our definition of smart, security aware GUIs. After having introduced the formal concepts for our security transformation, we apply the same methodology but refined to obtain smart, security-aware GUI models. We construct functions which compute the access-control information needed to annotate GUI's widgets according to the underlying security policy holding on the data model and attending to the structure of the GUI design. We show how these functions correctly fulfill our definition and help us to specify our QVT smart transformation which is also described in this chapter.

Chapter 6: Conclusions and Future Work. In this chapter we highlight the benefits and limitations of our approach and raise several questions that are opened for future work.

Chapter 2

Background

2.1 Overview

This chapter provides an introductory overview of the technologies and concepts used in this work in order to give the reader the needed background information for understanding the methodologies used in later chapters. We describe model-driven security in general and in this context how the SecureUML language can be combined with different design languages. Also, we introduce the OCL language that we use later on to analyze models. Finally, we introduce the concepts of model transformations using MOF QVT and a case study that will serve as a guiding example for the remaining chapters.

2.2 Object Constraint Language

Modeling, especially software modeling, has traditionally been a synonym for producing diagrams. Most models consist of a number of "bubbles and arrows" pictures and some accompanying text. The information conveyed by such a model has a tendency to be incomplete, informal, imprecise, and sometimes even inconsistent. Many of the flaws in the model are caused by the limitations of the diagrams being used. A diagram simply cannot express the statements that should be part of a thorough specification.
The UML notation is largely based on diagrams. However, to provide the level of conciseness and expressiveness that is required for certain aspects of a design, the UML standard defines the Object Constraint Language (OCL) [16]. OCL is a textual language with a notational style similar to common object oriented languages. Currently, this language is used for defining queries, referencing values, or stating conditions and business rules in a model.
OCL expressions are declarative and side effect-free. Specifically, OCL supports the expression of the modeler to specify precise and detailed constraints on the behavior of a model, without getting embroiled in implementation detail. An essential characteristic of OCL is that it is a strongly typed language. However, OCL adopts a simple non-symbolic syntax and restricts itself to a small set of core concepts. It is designed for usability: "it should be easily read and written by all practitioners of object technology and by their customers, i.e., people who are not mathematicians or computer scientists." [27].

The language includes predefined types like `Boolean`, `Integer`, and `String`, with standard operators like **not** and **or** over `Boolean`, $+$, and $*$ over `Integer`, and **substr** and `concat` over `String`. For example, $2 + 5$ and **not**$(2 + 5 = 6)$ are OCL expressions of type `Integer` and `Boolean`, respectively. The language also provides operators for generating collection types from more basic types, along with standard operations on collections like **union**, `includes`, or **size**. For example, **Set**(`Integer`) is the type of sets of integers and **Set**{1, 4, 6}−>**union**(**Set**{3}) is an expression of type **Set**(`Integer`) that denotes the union of the sets $\{1, 4, 6\}$ and $\{3\}$; **Set**{1, 4, 6}−>`includes`(3) is an expression of type `Boolean` that denotes the result of checking whether 3 is included in the set $\{1, 4, 6\}$; and, finally, **Set**{1, 4, 6}−>**size**() is an expression of type `Integer` that denotes the size of the set $\{1, 4, 6\}$. Iterator operators like `forAll`, **select**, or `collect`, operate on collection types. Each takes an OCL expression as an argument and specifies an operation computed over the elements of a collection. For example, **Set**{1, 4, 6}−>`forAll`(i|i > 7) is an expression of type `Boolean` that evaluates to **true** if and only if each element of the set $\{1, 4, 6\}$ is greater than 7.
The OCL language is open in the sense that it is parametric. Expressions are written in the context of a UML model, using the types and vocabulary provided by the model. The new types correspond to the classes in the model and the new vocabulary correspond to the properties (attributes, roles, and operations) declared for these classes. For example, consider a class diagram M containing a class A. Suppose too that this class has an attribute x of type `String`. Now, x can appear in OCL expressions, using dot notation: for an object o of the class A, the expression $o.x$ denotes the value of its attribute x.
OCL provides a convenient shorthand notation for navigating over multiple association ends: Whenever a property call (attribute, operation, or association end call) is applied to a collection, it will be interpreted as a `collect` over the members of the collection with the specified property. OCL also provides access to the value of certain properties of the classes themselves using the dot notation. For example, the expression A.`allInstances`() denotes the set of all objects of the class A.
In the following chapters, we use OCL to define constraints on our models and model transformations, to validate whether different properties hold on them and to define operations that are used within our MOF QVT transformations.

2.3 Model-Driven Security

Model-driven security [3] is a specialization of model-driven development to 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. This model-centric development approach is centered around the construction and analysis of *security-design models*, which are models that combine security requirements with system designs.

2.3.1 SecureUML language

SecureUML is a modeling language based on RBAC [9] for formalizing access control policies on protected resources [3]. 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 2.1 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*.

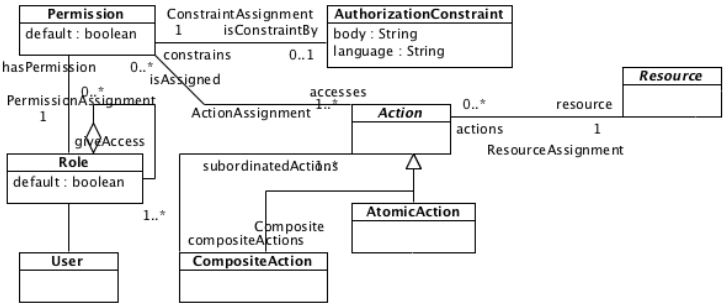

Figure 2.1: The SecureUML metamodel.

2.3.2 ComponentUML

ComponentUML is a simple language for modeling component-based systems. Its metamodel is shown in figure 2.2. Essentially, it provides a subset of UML class models: *Entities* can be related by *Associations* and may have *Attributes* and *Methods*. In this work, we use ComponentUML to model the application-data models on which our security models and GUI models are based.[1]

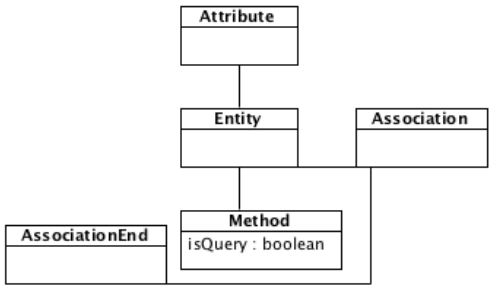

Figure 2.2: The ComponentUML metamodel.

[1]Note that the methodologies proposed in this work do not depend on this specific kind of data modeling language. Since ComponentUML is a simple language that appears in several scientific work in the context of model-driven security for modeling the application data, we decided to use this modeling language as well.

2.3.3 SecureUML+ComponentUML

The SecureUML+ComponentUML metamodel, shown in Figure 2.3, 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 and methods and entity creation and deletion.

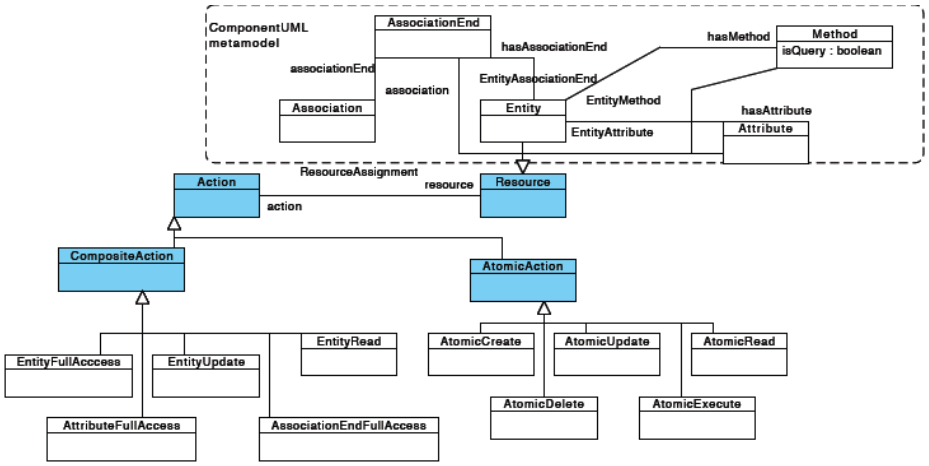

Figure 2.3: The SecureUML+ComponentUML metamodel.

In [3] 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. The authorization constraint expressions are attached to the permissions' association classes. ComponentUML entities are represented by UML classes with 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.

Abstract vs. Concrete Syntax When UML first appeared its definition was provided mainly by the graphical elements that could be used in the models (*notation*) together with a description of their intended meaning. Currently, the semantics of UML modeling elements is provided by a more precise conceptual model that is called the *metamodel* of UML. The instances of this metamodel that fulfill the well-formedness constraints imposed on it are UML models that are *correct*, i.e., well constructed, in what is called the *abstract syntax*. One or several metamodeling elements are used to provide the meaning of a graphical modeling element in *concrete syntax*. The advantage of graphical modeling elements is that they provide a summarizing form of expressing the meaning of a design, however for this very reason, the exact semantics that they convey must be found in their corresponding defining elements in abstract syntax.

Before we have described the UML profile introduced in [3] that is proposed for modeling security design models. The authors propose this notion as the concrete syntax. The instances of the profile, e.g. in figure 2.4, do hold the access-control information partly explicitly but partly implicitly w.r.t. to the semantics of a security design model. Its complete semantics in this case is defined in abstract syntax by its corresponding SecureUML+ComponentUML metamodel instance that holds all the access-control information explicitly.

SecureUML+ComponentUML example. Consider a basic PhoneBook application. 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 write phone numbers.
- Administrators are allowed to create and delete entries in the phone directory, as well as to read and write them.

Figure 2.4 shows the SecureUML+ComponentUML model that specifies the above policy.

2.3.4 Analyzing SecureUML models

In [1] the authors show how to automate the analysis of SecureUML+ComponentUML models in a semantically precise and meaningful way. More concretely,

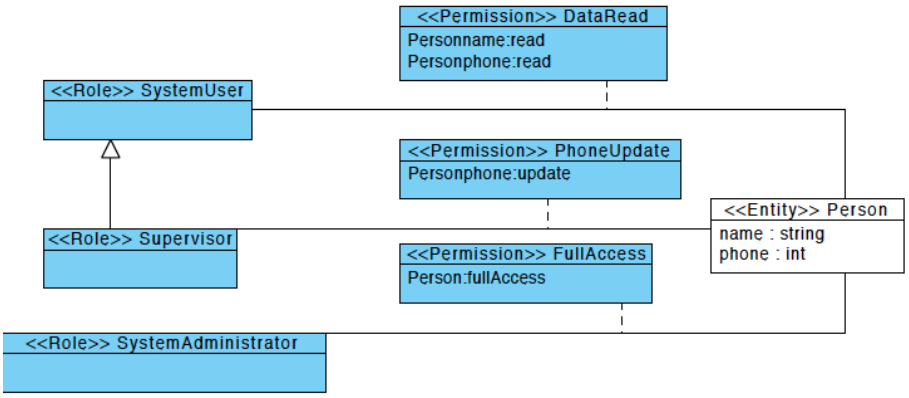

Figure 2.4: A simple security policy for a PhoneBook application.

they show that security properties of security-design models can be expressed as formulas in OCL [16], the Object Constraint Language of UML, in the context of the SecureUML+ComponentUML metamodel. Using this approach, we can formalize queries about the relationships between users, roles, permissions, and actions, and we can answer these queries by evaluating them on the instances of the SecureUML+ComponentUML metamodel that represent the security-design model under consideration. An example of a typical query about a security policy is "are there two roles such that one includes the set of actions of the others, but the roles are not related in the role hierarchy?". In [1] a number of OCL operators, in the context of the metamodel of SecureUML+ComponentUML, that formalize different aspects of the access control information contained in the security-design models are defined. In this work we will make use of these OCL operators and extend the list of SecureUML analysis operations.

Analysis operations In this work we mainly use the following functions that have been defined making use of the operators declared in [1][2].

Given action `self`, the function `DaAu()` returns the set of roles that can perform the action.

```
context Action::DaAu():Set(Role)
body: Role.allInstances()->select(r | r.allPermissions().accesses.
  subactionPlus()->includesAll(self.subactionPlus()))
```

Given an action `self` and a role `rl`, the function `AuthConst(rl:Role)` returns the authorization constraint constraining the permission of `rl` on `self`.

```
context Action::AuthConstA(rl:Role):String
body: setDisjunction(
  Permission.allInstances()->select(p | p.accesses.subactionPlus()->
    includes(self) and rl.allPermissions()->includes(p)).
    isConstraintBy._body->asSet())
```

[2]The OCL operators that are re-used below are explained in detail in [1] and are full-defined in the appendix.

2.4 Model Transformation

The main technique that we will use in this work is the process of transforming several input models into an output model. A more formal definition of this process is the following:

Definition 1 (Model transformation)**.** *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) following a set of mapping rules $R_1, \ldots, R_s$. This process is also known as Model-to-Model transformation.*

Several technologies can be used to perform such transformations. For example one could use model representations in XML and transform these models directly using XSLT. However, as the Object Management Group (OMG) provides a specification [15] for the definition of Model-to-Model transformations on MOF compliant models , we decided to use QVT to define our model-to-model transformations. Also since OCL is an integrated part of QVT, this technology offers not only the capability of making transformation definitions but also the possibility of defining OCL queries to be evaluated on different models at the same time. This property of QVT eases the task of defining and validating the correctness and completeness of the proposed transformations. More specifically we are able to analyze the information preservation of our transformations as pre/post-condition checks.

2.4.1 MOF QVT

MOF QVT (Query/View/Transformation) [15] is a OMG specification to describe executable Model-to-Model transformations which is part of the Meta Object Facility (MOF) standard. QVT allows model transformations to be specified with either a declarative or an imperative language. Figure 2.5 visualizes the architecture of the specification.

Declarative Part. The declarative part consists of QVT Relations and the QVT Core, as well as of the image language RelationsToCoreTransformation.

Imperative Part. The imperative part defines Operational Mappings and an interface for Black Box implementations that use other transformation languages than QVT.

In the following chapters we will use operational mappings to describe our model transformations. Thus we further explain and exemplify this part of QVT in the following section.

Operational QVT

The description of syntax for operational mappings can be found in [15, chapter 8]. Next we reproduce a very simple example borrowed from the appendix of [15] to show the syntax and usage of operational QVT.

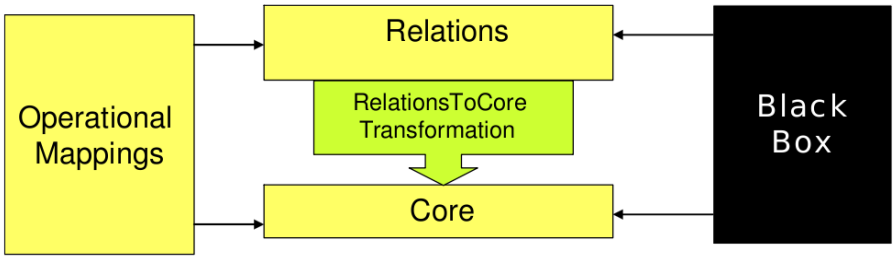

Figure 2.5: Overview QVT Architecture

Operational QVT transformation example.

This example shows the definition of a transformation from a `BOOK` model to a `PUB` model[3]. A `BOOK` model describes books, that consist of a title and several chapters. The chapters consist of a title and its number of pages. The `PUB` model describes publications, consisting of a title and the overall number of pages of the publication. The following transformation takes a `BOOK` input model and returns a `PUB` model as the output.

```
metamodel BOOK {
  class Book {title: String; composite chapters: Chapter;}
  class Chapter {title : String; nbPages : Integer;}
}

metamodel PUB {
  class Publication {title : String; nbPages : Integer;}
}

transformation Book2Publication(in bookModel:BOOK, out pubModel:PUB)
main() {
  bookModel->objectOfType(Book)->map book_to_publication();
}

mapping Class::book_to_publication () : Publication {
  title := self.title;
  nbPages := self.chapters->nbPages->sum();
}
```

2.4.2 Eclipse QVTO

Although there are several implementations of QVT in different platforms, we have chosen the operational QVT implementation, i.e., QVTO, that the Eclipse platform offers within the M2M-project since we know that the evaluation of OCL statements performed by this platform is one of the fastest even on large scenarios [5].
The M2M-project [24] is a subproject of the Eclipse Modeling project and it delivers a framework for Model-to-Model transformation languages. The core part

[3]Notice that QVT concrete implementations perform the transformation on models written in abstract syntax

is the transformation infrastructure. Transformations are executed by transformation engines that are plugged into the infrastructure. There are three transformation engines that are being developed within the M2M-project. Each of them represents a different category, which validates the functionality of the infrastructure for multiple contexts. The transformation engines provided by the M2M-project are ATL, Procedural QVT (Operational) and Declarative QVT (Core and Relational). The syntax of QVTO used by the M2M-project slightly differs from the syntax provided by the QVT standard but since the differences are really small, we will provide in the following chapters the transformation definitions just in QVTO syntax.
We specify the transformations described in chapters 4 and 5 to be executed by the eclipse platform as a proof of concept. The detailed definitions and some examples can be found in appendixes A, B and C.

Metamodel description. Metamodels are described as Ecore models inside the Eclipse Modeling Framework (EMF). EMF offers convenient tool support to create plug-ins and editors from Ecore models. In particular, some of these plug-ins and editors serve to instantiate the ecore (meta-)models to create sample input models for testing purposes. The reader will find a simple informal how-to in appendix D.

QVTO transformation example. Next we show the concrete implementation in QVTO of the Book-to-Publication transformation example given in section 2.4.1. This implementation is provided with [8]. We assume here the metamodels already implemented as ecore models.

```
modeltype BOOK uses 'http://book/1.0';
modeltype PUB uses 'http://pub/1.0';

transformation Book2Publication(in book : BOOK, out publication : PUB);

main() {
  book.objectsOfType(Book)->map book_to_pub();
}

mapping BOOK::Book::book_to_pub() : PUB::Publication
{
  title := self.title;
  nbPages := self.chapters->nbPages->sum();
}
```

2.5 Case Study

In Figure 1.1 we already depicted a simple example. Here we extend this example and define it in more detail to use it as a guiding example through the following chapters. Our example considers an application for managing information about a company's employees, i.e. editing and viewing employee information. The application data model depicted in figure 2.6 consists of a simple entity *Employee* with two attributes *name* and *id*.

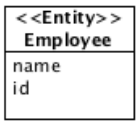

Figure 2.6: The simple data model in ComponentUML.

Security Policy We consider that three different kinds of users can interact with this application: *system users*, *members of the Human Resources (HR) department* and *system administrators*. The application shall provide *system users* with the possibility of viewing any employee information. *Members of the HR department* are allowed to read all employees' information and update it. Furthermore they can create new employees in the system but cannot delete any. Deletion of employees is a task reserved for *system administrators* only. *System administrators* have full access on employees' information. Figure 2.7 shows this access-control policy described with a SecureUML+ComponentUML model.

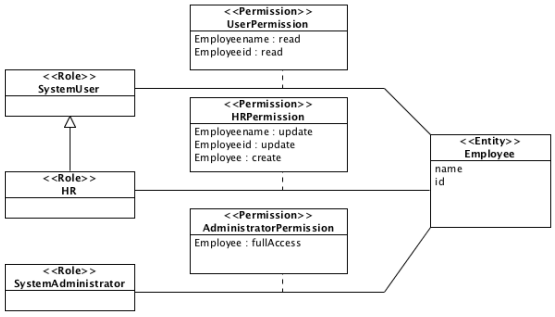

Figure 2.7: The security policy in SecureUML+ComponentUML.

GUI Design A simple GUI design[4] is depicted in figure 2.8 *a*). The GUI designer annotates the widget's events with the information of which data model actions are being triggered. Figure 2.8 *b*) shows the annotations made by the GUI designer in our case study, i.e. the behavioral part of the GUI design.

Smart, Security-Aware GUI model The methodologies explained in the coming chapters are aimed to automatically transform GUI models like the one provided above into smart, security-aware GUI models annotated with the corresponding access-control information for each widgets' event. Notice that these GUI drawings have first to be represented as a GUI metamodel instances if we want to apply the technologies presented before. However, even this small example becomes very large and hard to read in abstract syntax so we omit here such presentation. Instead we provide a CD attached to this work with this example as a ready input file for the implementations in QVTO of the transformations that we describe in the following chapters. In this CD those implementations of the QVT transformations can also be found together with a simple tutorial to get the reader easily started with the execution of the transformations. By the execution of the transformations on this file, the reader will obtain automatically the smart, security-aware GUI model.

[4]We introduce GUI design and the structure of GUIs in more detail in the next chapter.

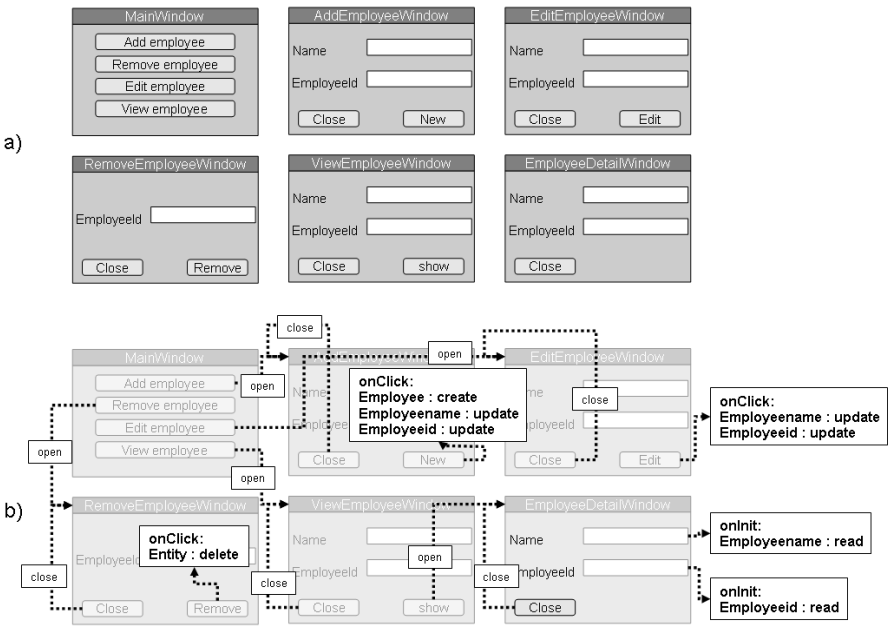

Figure 2.8: GUI design: a) Visual part, b) Behavioral part

Chapter 3

Modeling Secure GUIs

3.1 GUI design

In this section we summarize the main concepts of GUI design that we use in this work. For the sake of simplicity, we only consider those concepts that will play a relevant role in designing smart, security-aware GUIs.

A GUI Development Process There are many modeling languages supporting in the early stages of the development process of software applications the modeling of the application data while still few support exist to automatically generate the code of an application. On the contrary, there are numerous plug-ins and tools supporting the actual programming of a GUI but very few modeling languages to design it. Probably, this is the state of facts because very seldom, the design of a GUI for an application is included in its development process from the first phases. As a consequence of this, the process of building the data model and the one of designing GUIs often remain completely independent. A GUI designer usually uses a Rapid Application Development tool (RAD) to build a GUI model according to some predefined requirements, which later on is used to generate platform specific code fragments. In a further step the generated GUI code is bound manually to the corresponding data model actions at code level. This process impedes a GUI of being aware of an access control policy established for its application data till the moment the application development is already finishing. In fact, the task of making the GUI aware of such an access-control policy defined on the underlying data model is delegated to the GUI designer becoming obviously an error prone task. Here we propose an integrated development process for both application data and GUIs that in addition to other benefits, will bring the possibility of considering the access-control information during the modeling process of the GUI before producing any code. It will also release the GUI designer of the responsibility of making the GUI security aware when indeed he is probably not the security expert. Moreover we provide the GUI designer with a tool to automatically check that he has designed the right interface before actually generating any code.

Structure of a GUI In GUI design it is useful to distinguish between a GUI's visual elements and the behavioral properties associated with these elements. A GUI's visual elements are typically called *widgets*, which are of different types and which support different *events*. In this work, we consider the following widgets: *buttons*, *entries*, *containers*, and *windows*. Buttons are widgets that can be *clicked on*, e.g., the widget 'Close' in Figure 3.1. Entries are widgets that can be *filled in* with text, such as the widget 'EmployeeId' in Figure 3.1. Containers are widgets that can graphically

contain (group together) other widgets. Windows are a concrete class of containers, for example, in Figure 3.1 two windows are shown.

The behavioral properties of the different widgets are defined by the *actions* associated to the events that they support. It is useful to distinguish between *data actions* and *widget actions*. Data actions act on application data. Examples are *create*, *delete*, *read*, or *update* information in the application data. Widget actions act upon widgets (including themselves), like *open* or *close* a widget.

Finally, we distinguish between *compulsory* events and *non-compulsory* events. Compulsory events are events whose associated actions explain why their widgets are important *for* the GUI designer. For example, each widget-container supports, by default, a distinguished compulsory event whose associated actions are the actions of opening all its containee-widgets: visualizing together the containee-widgets is ultimately the reason for opening a container-window. Similarly, the event *onClick* of the button 'New' in Figure 3.1 is compulsory for its GUI designer: creating a new employee is the reason for showing this button in the corresponding window.

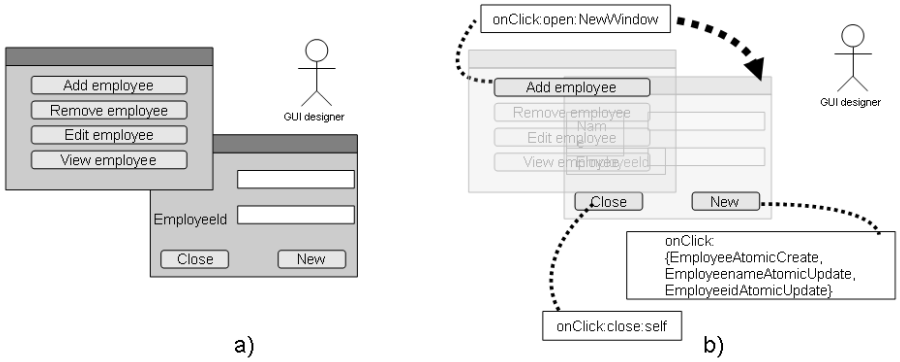

Figure 3.1: Smart, security-aware GUI modeling process

3.2 A GUI modeling language

In this section we describe a GUI modeling language by defining a MOF compliant metamodel to capture the concepts of the simple GUI structure which we have informally introduced above. We precise such metamodel using OCL invariants that have to be fulfilled by GUI models in order to be valid with respect to our definition of a GUI. We keep the modeling language quite simple in order to focus on the essential concepts. However, the language remains sophisticated enough to show our approach. At the end of this section we will talk about possible extensions that could be considered in future work. In figure 3.1 the modeling process of a GUI design is depicted. First *a*), the GUI designer creates the appearance of the GUI. Then *b*) he binds the widgets to concrete data model actions and widget actions by introducing the corresponding events.

3.2.1 GUI metamodel

In figure D.1 we depict our GUI metamodel. A GUI model consists of *Widgets* that can be of type either *Button*, *Entry* or *Container*. *Container* widgets can contain other widgets. In this work we just consider one kind of container, i.e. *Windows*.

Widgets can hold any number of events but each kind of *Widget* supports a specific set of possible events, e.g., the event *onEnter* only makes sense for an *entry* widget. Literal events are enumerated in the enumeration class *EventEnum*. *CompEvents*, as described above, are events whose associated actions must be fired because they are the reason why those widgets are important *for* the GUI designer. An event can fire several *GActions* that can be either *WidgetActions* acting on *Widgets*, i.e. *open* or *close*, or *ModelActions* acting on the data model, i.e. executing a data model action of type *AtomicAction*. This association of GUI events to data model actions is the key for what follows. For the sake of simplicity we consider only atomic data actions, however, we do not lose the generality of the approach since composite actions are at the end aggregations of atomic actions and we do not restrict either the multiplicity of the atomic data actions that are associated to an event. Therefore, since an *Event* can fire an arbitrary number of *GActions*, every composition of *AtomicActions* is possible in this model.[1]

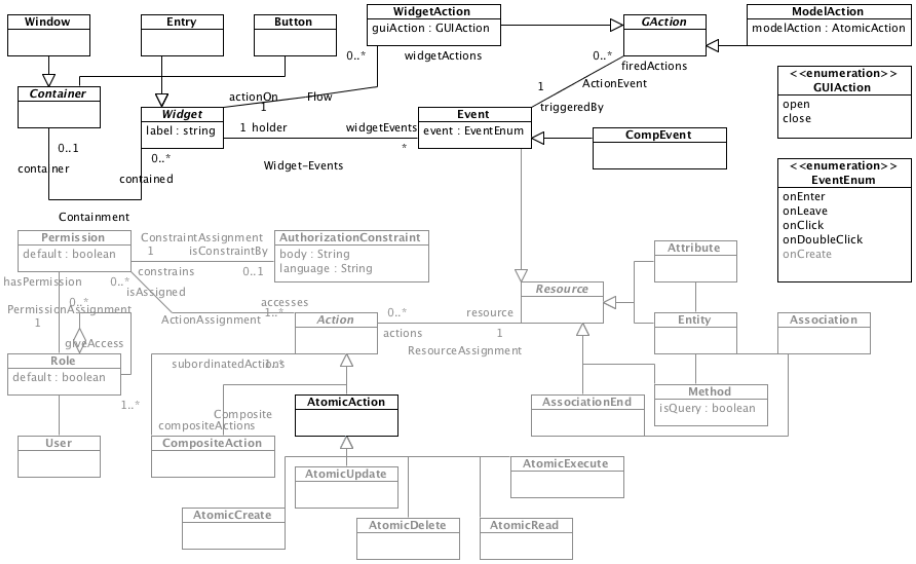

Figure 3.2: GUI metamodel

3.2.2 Constraints

It is known that modeling languages often lack the expressiveness that is necessary to capture all the requirements of a system design. In general, the OCL language is needed and is used to precise modeling languages; this is also the case of our GUI metamodel. Some of the requirements assumed above about the structure of a GUI have to be expressed using OCL invariants:

- *Widget*, *Container* and *GAction* are abstract metaclasses[2].

[1] Notice that we do not consider here the order of the actions. This is not important for the model, as the actual implementation on the code level remains as a task for the GUI designer. The methodology simply focuses on the access-control on events. How the implementation is finally performed is out of the scope of this work.

[2] The metaattribute *isAbstract: Boolean* means when true that the metaclass does not

```
context Widget inv:
self.isAbstract = true
context Container inv:
self.isAbstract = true
context GAction inv:
self.isAbstract = true
```

- Assignment of the supported events for the specific widgets.

```
context Window inv:
self.widgetEvents->isEmpty()
context Button inv:
self.widgetEvents->forAll(e |
e.event = EventEnum::onClick or
e.event = EventEnum::onDoubleClick)
context Entry inv:
self.widgetEvents->forAll(e |
e.event = EventEnum::onEnter or
e.event = EventEnum::onLeave)
```

- Each *ModelAction* references an *AtomicAction* on a ComponentUML resource.

```
context ModelAction inv:
self.modelAction.oclIsKindOf(AtomicAction)
context ModelAction inv:
self.modelAction.resource.oclIsTypeOf(Entity) or
self.modelAction.resource.oclIsTypeOf(Attribute) or
self.modelAction.resource.oclIsTypeOf(AssociationEnd) or
self.modelAction.resource.oclIsTypeOf(Method)
```

- Each kind of event exists at most once for each widget.

```
context Widget inv:
self.widgetEvents->forAll(e1, e2 | e1 <> e2 implies
e1.event <> e2.event)
```

3.2.3 Extensibility

In this work we only provide a simple GUI metamodel focused on the basic concepts that play distinct roles in a GUI. However, we keep in mind the need for some extensions in order to provide a language to model more complex GUIs. In this section we provide a short informal description of how these extensions can be carried out. Shortly speaking, our GUI metamodel can be extended in three ways, i.e., by the inclusion of new widgets that can be either containers or not, by the inclusion of new events that can be either compulsory or not and and by the inclusion of new actions that can be either model actions or widget actions. In general, we can say that the extensions can be carried out attending to the type of the elements to be inserted.

Introducing new elements. In most applications there exist many more types of widgets like *comboboxes*, *textareas* and the like. To include more types of widgets in our metamodel we simply introduce the new widget type either as a subtype of *Widget*

provide a complete declaration and can typically not be instantiated. An abstract metaclass is intended to be used by other metaclasses. Default value is false.

if it is a non container widget or as a subtype of *Container* if it is a container widget. If new events are to be included, for example, *onDestroy*, it is sufficient to insert this new literal in the enumeration class *EventEnum*. If new GUI actions are to be included, for example *hide*, it is sufficient to insert this new literal in the enumeration class *GUIAction*. The inclusion of new model actions depends on the extension of the actions that can be performed on the data model.
Finally, the OCL expressions have to be reconsidered, for example, those constraining the supported events for a new widget:

```
context NewWidget inv:
self.widgetEvents->forAll(e | e.event = EventEnum::newEvent or ...)
```

or those establishing that a new event should always be treated as a compulsory event:

```
context Event inv:
self.event = EventEnum::onInit implies self.oclIsTypeOf(CompEvent)
```

Using this method seems to be enough for most of the usual extensions. However a further study is left opened for future work.

Example of extensions: Considering the initialization of widgets. Often widgets are initialized with some data. For example, when opening text entries, they can hold some text preloaded. This information could be retrieved as the result of a read action on the data model. We do not consider this case explicitly in our examples but with a little modification of our metamodel it would be directly considered in our definitions and methodology without more additional changes:
Just a new kind of event, e.g. *onInit*, should be inserted into the enumeration class *EventEnum*. The idea is that whenever a widget holding such an *onInit* event is being showed, the assigned actions are being fired while loading the widget in order to initialize itself. Therefore only users allowed to perform this *onInit* event can see the corresponding widget. Thus, this *onInit* event has to be always a compulsory event in our model. More importantly, the OCL constraints for all widgets able to support this *onInit* event have to be altered accordingly. For example those defined on the text entries together with the constraint shown above to declare compulsory an event.

```
context Entry inv:
self.widgetEvents->forAll(e |
e.event = EventEnum::onEnter or
e.event = EventEnum::onLeave or
e.event = EventEnum::onInit)
```

3.3 Smart, Security-Aware GUI modeling

3.3.1 Concept

The basic idea for modeling smart, security-aware GUIs is to take the security-aware data model defined by the security engineer and the GUI model defined by the GUI designer and automatically aggregate them into one combined security-model[3]. This security model holds all the information needed to compute the access-control information to annotate the GUI events. As we make use of the approaches taken in [1] to automatically analyze SecureUML models using OCL, the security policy may include

[3]We provide an implementation of this aggregation using operational QVT in Appendix A. In this implementation we have also defined QVT queries to check that the information is preserved by the transformation.

both declarative aspects, i.e., static access control information such as the assignment of users and permissions to roles, and programmatic aspects, which depend on dynamic information, namely the satisfaction of authorization constraints in the given scenario.
At the end, we would like to have a concrete running implementation of the ideas presented here. In such application there should exist a mechanism for getting this contextual information enabling the analysis somehow.[4]

3.3.2 Combined Security Model

Combined security models are obtained by the aggregation of a SecureUML+ComponentUML model and a GUI model and it conveys all the information contained in both models. In the security combined metamodel we are able to automatically analyze using OCL queries the dependencies between the GUI elements and the data model actions. In addition, we can compute the access-control information to annotate the GUI events and obtain a SecureUML+GUI model where these events are the protected resources.

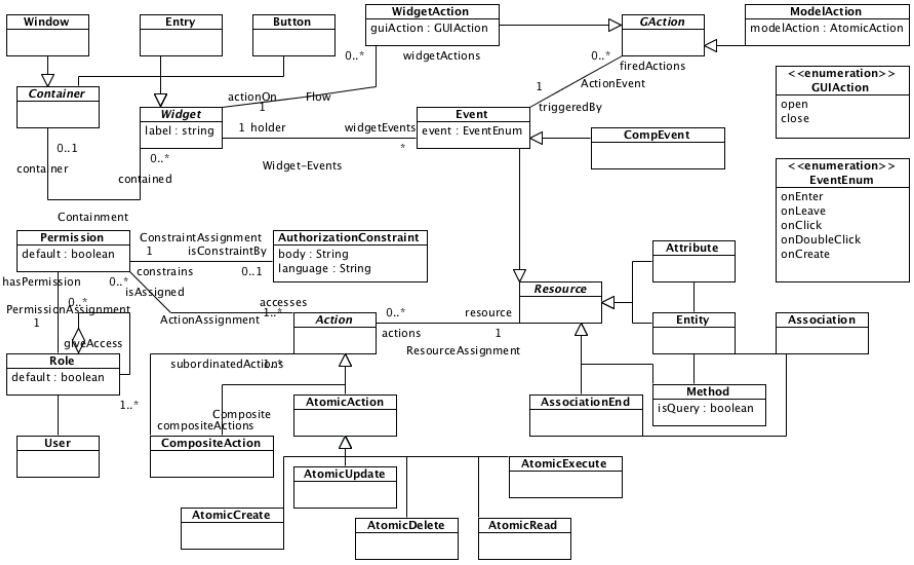

Figure 3.3: Combined metamodel

3.3.3 Constraints

The following constraints have to hold on any combined security model. They are similar to the constraints established for both GUI models and SecureUML+ComponentUML models but with some slight modifications, for example every widget must hold exactly one onCreate event. These onCreate events are further constrained depending on whether the corresponding Widget is a Container or not. The exact explanation of these constraints will become clear in the following chapters. By now it is simply important that after the aggregation step the following properties hold.

- Every widget holds an *onCreate* event.

[4] In this work, we focus on the theoretical aspects of transforming the security models correctly. The problem of the actual code generation is left open for future work.

```
context Widget inv:
self.widgetEvents->exists(e | e.event = EventEnum::onCreate)
```

- Every *onCreate* event is of type *CompEvent*.

```
context Event inv:
self.event = EventEnum::onCreate implies
self.oclIsTypeOf(CompEvent)
```

- For non-containers there is no action assigned to *onCreate* events.

```
context Widget inv:
not self.oclIsKindOf(Container) implies self.widgetEvents->
select(e | e.event = EventEnum::onCreate).firedActions->isEmpty()
```

- Only *open* widget actions are fired by *onCreate* events.

```
context Event inv:
self.event = EventEnum::onCreate implies self.firedActions->
forAll(a | a.oclIsTypeOf(WidgetActions) and
a.oclAsType(WidgetAction).guiAction = GUIAction::open)
```

- The *onCreate* event of a container is assigned to one *open* widget action for every widget that it contains.

```
context Container inv:
self.widgetEvents->select(e | e.event =
EventEnum::onCreate).firedActions->
select(a | a.oclIsTypeOf(WidgetAction))->
collect(b | b.oclAsType(WidgetAction))->asSet() =
self.contained.widgetActions->
select(a | a.guiAction=GUIAction::open)->asSet()
```

- Every event has exactly one *AtomicExecute* action defined on it.

```
context Event inv:
self.actions->size = 1 and self.actions->asOrderedSet()->
first().oclIsTypeOf(AtomicExecute)
```

Note that the constraints stated in [1] apply to the SecureUML+ComponentUML part but not to the SecureUML+GUI part. For example, every action in the SecureUML+ComponentUML model have to be accessed by at least one permission either by an explicitly defined permission or by the default permission. In the SecureUML+GUI part there might exist events with no permission defined on it at all, for instance, the situation where the GUI designer defined an event associated to a set of actions that no one is allowed to perform. This would be a fault of the GUI designer, but as we propose this methodology as well to verify the correctness of the defined GUI design, we consider this situation as well. Therefore, whenever an event has no permission assigned on its *AtomicExecute* action, this means, that no one is allowed to execute this event. More precisely, access on an event is granted only when it is explicitly declared. No default allowing permission is declared for the events.

3.3.4 SecureUML+GUI language

The SecureUML+GUI language is simply an integrated part of the combined security model explained above after the permission lifting transformation of permissions defined on the SecureUML+ComponentUML part to permissions defined on the SecureUML+GUI part. Figure D.3 shows this modeling language as part of the combined security metamodel highlighted in black. It is obvious that one could extract the SecureUML+GUI part easily from the combined security model by a simple model transformation using QVT.

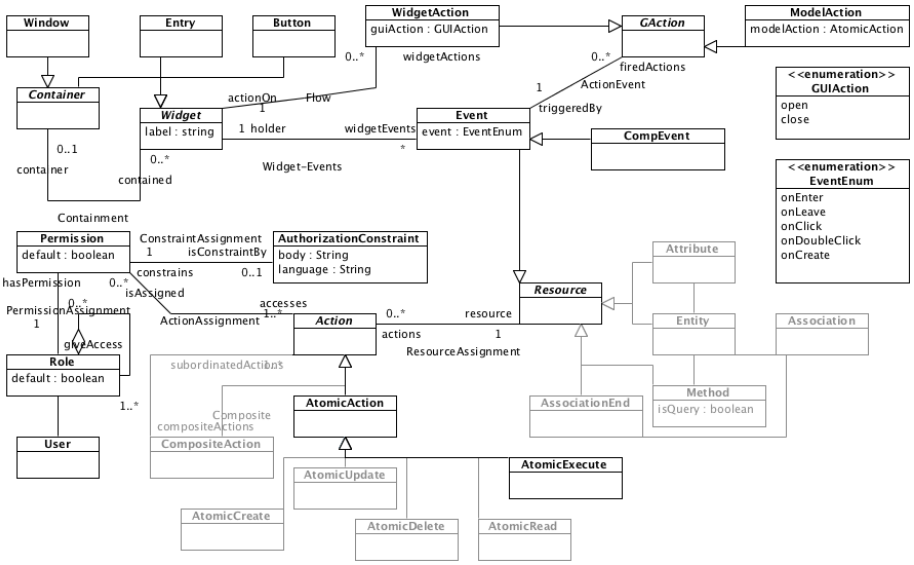

Figure 3.4: SecureUML+GUI metamodel part

The GUI metamodel is combined with the SecureUML metamodel using a *dialect* that defines *Event* as the only protected resource and *AtomicExecute* as the only action on the protected resources. Since we already defined the constraints on the combined security metamodel, the needed properties are defined, such that we can state OCL queries to analyze the corresponding models. Further constraints will be stated as post-conditions of the security transformations defined in the following chapters.

3.3.5 Analysis of SecureUML+GUI models

Again we make use of the metamodel-based methodology presented in [1] to automatically check properties on SecureUML+GUI models. In particular, we can formalize in OCL queries about the smart, security-aware properties of GUI designs. These queries are evaluated on the instances of the SecureUML+GUI metamodel that represent the GUI models under consideration. More generally, we can define different OCL operators in the context of the metamodel of SecureUML+GUI to analyze SecureUML+GUI models. For example,

- The function `Ev()` returns the set of events associated to a widget.

  context `Widget::Ev():Set(Event)`
  body: `self.widgetEvents`

- The function `CompEv()` returns the set of compulsory events associated to a widget.

```
context Widget::CompEv():Set(CompEvent)
body: self.Ev()->select(e|e.oclIsTypeOf(CompEvent))
```

- The function **In()** returns the widgets that are contained by a container.

```
context Container::In():Set(Widget)
body: self.contained
```

- The function `WdAc()` returns the set of widget actions that are triggered by an event.

```
context Event::WdAc():Set(WidgetAction)
body: self.firedAction->select(a|a.oclIsTypeOf(WidgetAction))
```

- The function `DaAc()` returns the set of data actions that are triggered by an event.

```
context Event::DaAc():Set(ModelAction)
body: self.firedAction->select(a|a.oclIsTypeOf(ModelAction))
```

- The function `Ac()` returns the set of actions (both widget and data actions) that are triggered by an event.

```
context Event::Ac():Set(GAction)
body: self.DaAc().oclAsType(GAction).union(
self.WdAc().oclAsType(GAction))
```

- The function `EvAu()` returns the set of roles which are allowed to execute an event.

```
context Event::EvAu():Set(Role)
body: Role.allInstances()->select(r | r.allPermissions().accesses
->includesAll(self.actions))
```

- The function `AuthConstG(rl:`**Role**`)()` returns the authorization constraint that a role with granted access to execute an event have in addition to fulfill to actually be able to execute it.

```
context Event::AuthConstG(rl:Role):String
body: self.actions.isAssigned->select(p | p.givesAccess = rl)
->asOrderedSet()->first().isConstraintBy._body
```

Now, we can reuse these operators to formulate other analysis questions, for example:

- Are there any events which denied access to everybody? Which are they?

```
context areTherenonExecutableEvents():Boolean
body: Event.allInstances()->exists(e | e.EvAu()
->isEmpty())
context getnonExecutableEvents():Set(Event)
body: Event.allInstances()->select(e | e.EvAu()
->isEmpty())
```

- Which are the events that trigger the same set of data model actions than the given one?

```
context Event::getEventsWithSameDataActionSet():Set(Event)
body: Event.allInstances()->select(e | e.DaAc().modelAction
= self.DaAc().modelAction)
```

- Which are the events that the given role is allowed to execute?

```
context Role::getEventsForRole():Set(Event)
body: Event.allInstances()->select(e | e.EvAu()->includes(self))
```

- Which data model actions is the given role allowed to execute through the GUI model?

```
context Role::getDataModelActionsForRole():Set(AtomicAction)
body: self.getEventsForRole().DaAc().modelAction->asSet()
```

As we have shown, many questions can be formulated in this setting to analyze SecureUML+GUI models.

3.4 Formalization of SecureUML+GUI models

In order to define what do the concepts of security-awareness and smartness mean on a GUI design and show the correctness of our approach, we next formalize the concepts of our GUI modeling language.

Given a widget wd, we denote by $Ev(wd)$ and by $CompEv(wd) \subseteq Ev(wd)$, respectively, the set of events and the set of *compulsory* events associated with the widget wd.
Also, given a container-widget wd, we denote by $In(wd)$ the set of widgets (directly) contained in wd.
Moreover, given an event ev, we denote by $DaAc(ev)$ and by $WdAc(ev)$, respectively, the sets of application-data actions and widget actions associated to the event ev.
Finally, we denote by $Ac(ev)$ the total set of actions associated to an event ev, i.e., $Ac(ev) = DaAc(ev) \cup WdAc(ev)$.

In what follows, we assume that there are only two kind of widget actions, namely, *open* and *close*
Further on we assume that every widget wd has a distinguished event $onCreate \in CompEv(wd)$, which satisfies the following property:

- If wd is a non-container widget, then

$$Ac(\text{onCreate}_{wd}) = \emptyset.$$

- If wd is a container widget, with $In(wd) = \{wd_1, \ldots, wd_n\}$, then

$$Ac(\text{onCreate}_{wd}) = \{(\text{open}, wd_1), \ldots, (\text{open}, wd_n)\}.$$

Now, given a graphical user interface $\mathcal{G}$ for an application $\mathcal{A}$, let $EvAu : Event \to Roles$ be the function that, given an event ev, returns the set of roles $\{rl_1, \ldots, rl_n\}$ that can execute the event ev.

Let $AuthConst_{\mathcal{G}} : Role \times Event \to AuthorizationConstraint$ be the function that given an event ev of the GUI model and a role rl with $rl \in EvAu(ev)$, returns the authorization constraint constraining the permission of rl on ev.

These functions formalize the concepts provided above for defining smart, security-aware GUI designs. The correspondence is shown in Table 3.1.

it is satisfied	it evaluates to **true**
$ev \in Ev(wd)$	wd.`Ev()->includes(`ev`)`
$ev \in CompEv(wd)$	wd.`CompEv()->includes(`ev`)`
$wd \in In(wd')$	wd'.**In**`()->includes(`wd`)`
$wdac \in WdAc(ev)$	ev.`WdAc()->includes(`$wdac$`)`
$daac \in DaAc(ev)$	ev.`DaAc().modelAction->includes(`$daac$`)`
$ac \in Ac(ev)$	ev.`Ac()->includes(`ac`)`
$rl \in EvAu(ev)$	ev.`EvAu()->includes(`rl`)`
$ac = AuthConst_{\mathcal{G}}(rl, ev)$	ev.`AuthConstG(`rl`)` $= ac$

Table 3.1: Correspondence between the OCL operations and formal concepts

Chapter 4

Security-Awareness

4.1 Introduction

The problem that we address in this chapter is how to automatically generate an application security-aware GUI model from the security model and the GUI model. The process of designing a security-aware GUI has the following parts:

1. Software engineers specify the application data model A.
2. Security engineers specify, in the security model $S(A)$, the application data access control policy[1].
3. GUI designers specify the application GUI model $G(A)$. This includes the definition of which events trigger which data model actions.
4. Finally, the application security-aware GUI model $S(G(A))$ is automatically being generated from the security model $S(A)$ and the GUI model $G(A)$.

Figure 4.1 depicts the security transformation we discuss further on in more detail.

In section 4.2 we provide our considerations of what a security-aware GUI for an application should be. Then, in section 4.3 we grasp the meaning of the previous considerations to provide a formal definition of what we understand by a security-aware GUI model. Since it is not reasonable to expect the GUI designer to annotate each GUI's widget with the information about who can execute its events, in section 4.4 we build two functions that automatically carry out such a task, i.e., $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$. In section 4.5 we show how these functions can be constructed in OCL. In section 4.6 we use these functions to specify the security transformation in QVT that automatically builds a security-aware GUI model from a security model and the GUI model. Finally, in section 4.7 we talk about the correctness of this security transformation.

4.2 Security-Awareness: Informal description

Informally we can say that a GUI design is security-aware if the two following statements hold:

- Whenever a user is allowed to perform an event in the GUI model, then this user is as well allowed to perform all the data model actions assigned to this event.

[1] Note that we consider here the security data model in abstract syntax including all information explicitly.

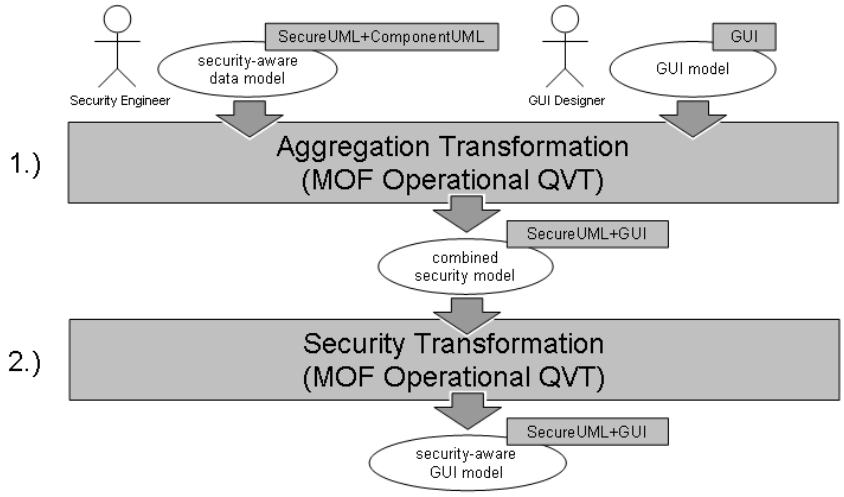

Figure 4.1: Generating security-aware application GUIs.

- Whenever a user has the permissions to perform the whole set of data model actions assigned to an event, then the user has as well the permission to execute this event.

Figure 4.2 shows a simple example of a GUI consisting of two windows. The first window includes several buttons, button *Add employee* has been linked to an *onClick* event that fires an *open* widget action opening the second window. The second window includes two text entries and two buttons. Button *New* has been linked to an *onClick* event that fires several model actions. The security-policy defined by the security engineer defines the set of roles R_i that is allowed to execute the corresponding model actions on the data model.

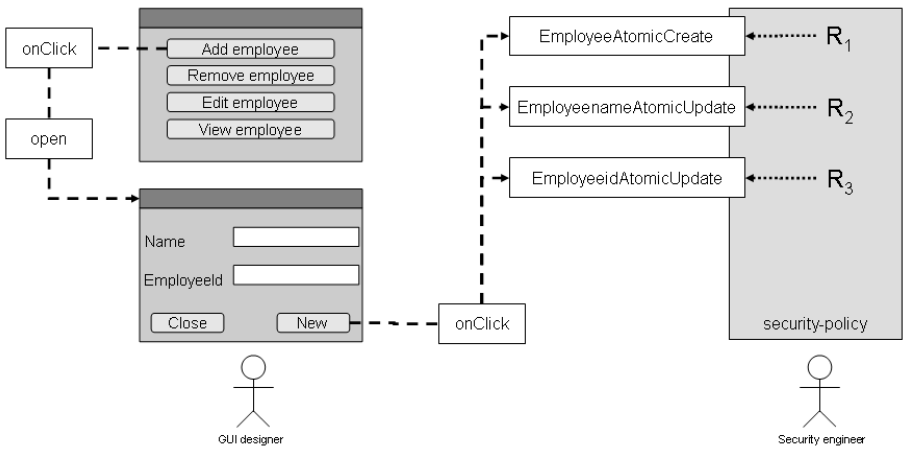

Figure 4.2: An example GUI before the Security Transformation.

The Security Transformation is now depicted in figure 4.3. The set of roles that is allowed to execute every model action assigned to an event is being computed,

i.e. the intersection of the role sets. This set of roles is being permitted now to execute the corresponding event. More specifically, for every role that is allowed to perform every model action assigned to an event, a permission is being generated granting the role access on the *AtomicExecute* action that is assigned to this event. From that role's permissions granting access to the model actions assigned to the event, the authorization constraints are collected and joined by conjunction in just one authorization constraint that is attached to the newly generated permission to execute the event. For every event that has no data model action assigned to it, it makes sense to allow access to everybody, i.e. for every role that is defined in the security data model, a permission is being created.

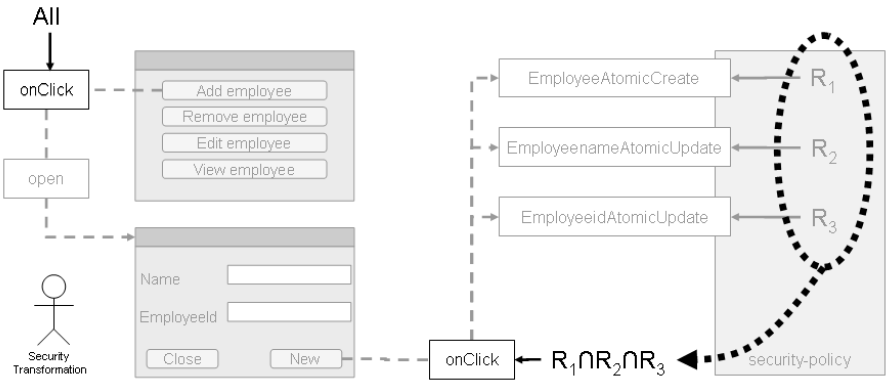

Figure 4.3: An example GUI during the Security Transformation.

4.3 Security-Awareness: Formal definition

Recall that we already provided a formalization for GUIs in section 3.4.

Further on we assume, there is a function $AuthConst_{\mathcal{A}} : Role \times DataAction \rightarrow AuthorizationConstraint$ that given an action $daac$ on application data and a role rl with $rl \in DaAu_{\mathcal{A}}(daac)$, returns the authorization constraint constraining the permission of rl on $daac$.

We are ready to formally define security aware GUI designs.

Definition 2 (Security-Awareness)**.** *Let $\mathcal{A}$ be an application governed by an access control policy. Then, a GUI design $\mathcal{G}$ for $\mathcal{A}$ is* security-aware *if and only if there exists a function $EvAu_{\mathcal{G}}$ and a function $AuthConst_{\mathcal{G}}$ such that for every event ev with $DaAc(ev) = \{daac_1, \ldots, daac_n\}$ the following two properties hold*[2]*:*

i. $EvAu_{\mathcal{G}}(ev) = \bigcap_{i=1}^{n} DaAu_{\mathcal{A}}(daac_i)$.

ii. $AuthConst_{\mathcal{G}}(rl, ev) = \bigwedge_{i=1}^{n} AuthConst_{\mathcal{A}}(rl, daac_i)$, *for each* $rl \in EvAu_{\mathcal{G}}(ev)$

[2]Note that security-awareness only focuses on events that are assigned to data model actions. It makes sense to assume here that when an event has not any data model action assigned, i.e. $DaAc(ev) = \emptyset$, it holds a permission such that every user has access on it without any constraint.

4.4 Construction of the functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$

Starting from a SecureUML+ComponentUML data model and a GUI model, one cannot expect the GUI designer to annotate each widget with the information about who can execute its events: first of all, because he may not be fully aware of the application data security policy, but also because this is a cumbersome and prone to error task, if done manually. We define two functions to carry out automatically such a task, we call these functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$. The former function is aimed to compute automatically for each GUI event which roles can execute it according to the permissions established on the data actions associated to the given event in the data model. The latter function is aimed to compute automatically for each role with granted access to execute an event under which constraints it is actually allowed to execute it.
In what follows we denote by 'All' the set of all the roles considered in an application-data security policy.

Remark 1. *Let $\mathcal{A}$ be an application protected by an access control policy. Then, given a GUI design $\mathcal{G}$ for $\mathcal{A}$ we can always construct a function $AllowedRoles_{\mathcal{G}}()$ that returns the set of roles $\{rl_1, \ldots, rl_n\}$ with permission on the given event ev w.r.t. the permissions established on the data actions associated to it in the data model.*

This construction is as follows:

- For every event ev with $DaAc(ev) = \emptyset$, then
$$AllowedRoles_{\mathcal{G}}(ev) := \text{All}\,.$$
- For every event ev with $DaAc(ev) = \{daac_1, \ldots, daac_n\}$, then
$$AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} DaAu_{\mathcal{A}}(daac_i)\,.$$

Remark 2. *Let $\mathcal{A}$ be an application protected by an access control policy. Then, given a GUI design $\mathcal{G}$ for $\mathcal{A}$ we can always construct a function that returns the authorization constraint constraining the permission on a given event ev for the given role $rl \in AllowedRoles_{\mathcal{G}}(ev)$ w.r.t. the constraints constraining the permissions for that role on the data actions associated to the event in the data model.*

The construction is as follows:

- For every event ev with $DaAc(ev) = \emptyset$, then
$$Constraint_{\mathcal{G}}(rl, ev) := \text{True}\,.$$
- For every event ev with $DaAc(ev) = \{daac_1, \ldots, daac_n\}$, then
$$Constraint_{\mathcal{G}}(rl, ev) := \bigwedge_{i=1}^{n} AuthConst_{\mathcal{A}}(rl, daac_i)\,.$$

4.5 $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$ in OCL

In chapter 3 we have already introduced our security GUI modeling language, i.e. SecureUML+GUI together with a list of operations for analyzing its models using OCL. In what follows we will refer to these definitions in order to implement the above formally described functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$ in OCL. As our model transformation later on is defined using operational QVT and OCL is part of QVT, we will use this implementation to transform the GUI model into a security-aware GUI model.

Implementation of function $AllowedRoles_{\mathcal{G}}$**.** The following operations implement the function $AllowedRoles_{\mathcal{G}}$ using OCL:

```
context Event::AllowedRolesG():Set(Role)
body: if self.DaAc()->isEmpty() then Role.allInstances()
  else setIntersection(
    self.DaAc().modelAction->collectNested(daac | daac.DaAu())->asSet()
  ) endif

context setIntersection(rs : Bag(Set(Role))):Set(Role)
body: if rs->size() = 1 then rs->asOrderedSet()->first()
  else rs->asOrderedSet()->first()->intersection(
    setIntersection(rs->excluding(rs->asOrderedSet()->first()))
  ) endif
```

Implementation of function $Constraint_{\mathcal{G}}$**.** The following operations implement the function $Constraint_{\mathcal{G}}$ using OCL:

```
context Event::ConstraintG(rl:Role):String
body: if self.DaAc()->isEmpty() then 'True'
  else setConjunction(
    self.DaAc().modelAction->collect(daac | daac.AuthConstA(rl))->asSet()
  ) endif

context AtomicAction::AuthConstA(rl:Role):String
body: setDisjunction(self.isAssigned->
  select(p | p.givesAccess = rl).isConstraintBy._body->asSet())

context setConjunction(acs : Set(String)):String
body: if acs->size() = 1 then acs->asOrderedSet()->first()
  else '('.concat(acs->asOrderedSet()->first()).concat(')␣and␣('.concat(
    setConjunction(acs->excluding(acs->asOrderedSet()->first()))
  ).concat(')') endif

context setDisjunction(acs : Set(String)):String
body: if acs->size() = 1 then acs->asOrderedSet()->first()
  else '('.concat(acs->asOrderedSet()->first()).concat(')␣or␣('.concat(
    setConjunction(acs->excluding(acs->asOrderedSet()->first()))
  ).concat(')') endif
```

Table 4.1 shows the correspondence between the formal definitions and the evaluation of OCL expressions on instances of the combined metamodel [3].

4.6 Security Transformation

In this section we describe the model transformation that automatically generates security-aware SecureUML+GUI models from SecureUML+ComponentUML models and GUI models. We assume here, as mentioned above, that the source models have the same ComponentUML application-data model. This automated transformation

[3]Rigorously proving this correspondence requires detailed meta-reasoning that involves both the semantics of the underlying formal system, the semantics of OCL, and the translation scheme from terms in the semantic domain to OCL expressions. This is a large undertaking and outside the scope of this thesis.

it is satisfied	it evaluates to **true**
$rl \in AllowedRoles_{\mathcal{G}}(ev)$	ev.`AllowedRolesG()->includes(`rl`)`
$ac = Constraint_{\mathcal{G}}(rl, ev)$	ev.`ConstraintG(`rl`)` $= ac$

Table 4.1: Mapping to OCL functions

carries out the last step of the high-level process description given in the introduction of the chapter. For simplicity we split the transformation into two sequential transformations:

1. Aggregation of the GUI model and the SecureUML+ComponentUML model into a single combined security model.
2. Permission lifting from the SecureUML+ComponentUML part of the model to the SecureUML+GUI part.

The first transformation takes as source models the SecureUML+ComponentUML model defined by the security engineer and the GUI model defined by the GUI designer and simply aggregates the information of these two models into one combined security model (see section 3.3.2). Listing 4.1 shows an excerpt of this QVT transformation.

Listing 4.1: The Aggregation Transformation in QVTO syntax (Excerpt).

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

transformation AggregationTransformation(in guiModel : GUI,
  in secModel : SECUMLANDCOMPUML, out guiSecModel : SECUMLANDGUI);

main() {
  secModel.objects()[SECUMLANDCOMPUML::User]->map User_to_User();
  secModel.objects()[SECUMLANDCOMPUML::Role]->map Role_to_Role();
  ...
}

mapping SECUMLANDCOMPUML::User::User_to_User() : SECUMLANDGUI::User
{
  name := self.name;
}

mapping SECUMLANDCOMPUML::Role::Role_to_Role() : SECUMLANDGUI::Role
{
  name := self.name;
  default := self.default;
  includes := self.includes.resolve()->oclAsType(SECUMLANDGUI::User)->
    asOrderedSet();
}
...
```

The second transformation takes the combined security model as the only input-/output model which is being transformed such that the permissions defined on the SecureUML+ComponentUML part are lifted to the SecureUML+GUI part according to the considerations made in the previous section. Listing 4.2 shows an excerpt of this QVT transformation.

Listing 4.2: The Security Transformation's heading in QVTO syntax.

```
modeltype SECUMLANDGUI uses "http://secumlandgui/1.0";

transformation SecurityTransformation(inout secGUI : SECUMLANDGUI);

main() {
  secGUI.objects()[Event]->map liftPermissions();
}

mapping Event::liftPermissions() : Set(Permission)
{
  init{
    result := self.AllowedRolesG()->setPermission(self)->asSet();
  }
}

mapping Role::setPermission(e: Event) : Permission
{
  name := self.name + e.holder.label + e.event.repr() + 'Execute';
  accesses := e.actions;
  default := false;
  givesAccess := self;
  isConstraintBy := self.map setAuthConst(e);
}

mapping Role::setAuthConst(e: Event) : AuthorizationConstraint
{
  name := self.name + e.holder.label + e.event.repr() + 'ExecutionConstraint';
  _body := e.ConstraintG(self);
  language := 'some language';
}
```

Next we describe these two transformations. Their full QVT specifications can be found in Appendix A and B.

Step 1: Aggregation Transformation.[4]

- Every element in the (source) SecureUML+ComponentUML model is copied, along with their hierarchies, into the (target) combined security model, using the corresponding mapping functions of the form:

  ```
  secModel.objects()[SECUMLANDCOMPUML::User]->map User_to_User();
  secModel.objects()[SECUMLANDCOMPUML::Role]->map Role_to_Role();
  ```

  ...

- Every element in the (source) GUI model is copied, along with their hierarchies, into the (target) combined security model, using the corresponding mapping functions of the same form as above.
- The links between the *ModelActions* of the GUI model part and the concrete data model *Actions* of the SecureUML+GUI model part are being established.

[4]Note that we defined QVT queries to check that the outcome of the aggregation transformation is sound according to our definitions of a secure GUI modeling language.

Step 2: Security Transformation.

- For each *Event* in the SecureUML+GUI model, and for each *Role* that is allowed to perform the *AtomicExecute* actions assigned to the *Event* according to the function *AllowedRolesG()*, a *Permission* is created in the SecureUML+GUI model that grants the role access to execute the event. The *Authorization-Constraint* constraining the newly created *Permission* is then being computed using function *ConstraintG()*[5]. The transformation is accomplished using the following mapping function:

```
secGUI.objects()[Event]->map liftPermissions();
```

Where `liftPermissions()` is defined as follows:

```
mapping Event::liftPermissions() : Set(Permission)
{
  init{
    result := self.AllowedRolesG()->setPermission(self)->asSet();
  }
}
```

The resulting combined security model can now be used for either code generation or to analyze the validity of the GUI that has been designed w.r.t. the requirements defined for the GUI as it is proposed in [1] using OCL. This information is crucial to the GUI designer in order to validate his GUI model, i.e., to check that he is designing the right graphical interface to give the authorized users access to the intended application data. At this point, he may realize that another GUI is needed to fulfill the intended purpose.

4.7 Correctness

We claim that our model transformation is correct, i.e. the GUI model obtained as an output of the transformation is security-aware w.r.t. definition 2. under the assumption of the correspondences between the formal functions and the corresponding OCL operations.

Theorem 1 (Correctness)**.** *Let $\mathcal{A}$ be an application protected by an access control policy. Let $\mathcal{G}$ be a GUI design for $\mathcal{A}$ obtained as the output model of the security transformation. Then, $\mathcal{G}$ is security-aware according to definition 2.*

Proof sketch. [6]

Remarks 1 and 2 show how the functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$ can be always constructed for every given GUI design $\mathcal{G}$. According to definition 2, by case distinction, we have to show that $EvAu_{\mathcal{G}}(ev) = AllowedRoles_{\mathcal{G}}(ev)$ and the function $AuthConst_{\mathcal{G}}(rl, ev) = Constraint_{\mathcal{G}}(rl, ev)$ to prove that $\mathcal{G}$ is security-aware. □

Therefore the outcome of our transformation is security-aware w.r.t. definition 2.
From the modeling point of view we can say that our transformation satisfies the expected property, namely, that the (target) SecureUML+GUI model *preserves* the security policy specified in the (source) SecureUML+ComponentUML model. More specifically, a user is allowed to execute an event in the (target) SecureUML+GUI model only if he is allowed, in the (source) SecureUML+ComponentUML model, to execute the actions that are associated to this event in the (source) GUI model.

[5] The full implementation of these functions using QVTO can be seen in Appendix B.
[6] Please, see the proof of this theorem in the next chapter, section 5.7.

Following we define the properties of definition 2 as invariants using OCL. These invariants can be used to check the correctness of the specific transformation output as a post-condition check.

```
context Event inv:
self.DaAc()->notEmpty() implies self.EvAu() = setIntersection(
self.DaAc()->collectNested(daac | daac.DaAu())
)

context Event inv:
self.DaAc()->notEmpty() implies self.EvAu()->forAll(rl |
  self.AuthConstG(rl) = setConjunction(
    self.DaAc()->AuthConstA(rl)
  )
)
```

In Appendix B we show an example where we check the above invariants along with other conditions on the SecureUML+GUI model that is the output of a security transformation.

Chapter 5

Smartness

5.1 Introduction

After defining security-aware GUIs in the last chapter. In this chapter we focus on *smartness*. The definition of smartness is much less clear than the definition of *security-awareness*. The smartness transformation described in this chapter extends the security transformation described in the previous chapter. The process to obtain a smart, security-aware GUI is the same process that to obtain a security-aware GUI:

1. Software engineers specify the application data model A.
2. Security engineers specify, in the security model $S(A)$, the application data access control policy[1].
3. GUI designers specify the application GUI model $G(A)$.
4. Finally, the application smart, security-aware GUI model $S(G(A))$ is automatically being generated from the security model $S(A)$ and the GUI model $G(A)$.

More concretely, the problem that we address in this chapter is how to automatically generate an application smart, security-aware GUI model from the application security model and the GUI model. First, in section 5.2, we provide our considerations of what a smart, security-aware GUI for an application should be. Then, in section 5.3, we grasp the meaning of the previous considerations to provide a formal definition of what we understand by a smart, security-aware GUI model. Since it is not reasonable to expect the GUI designer to annotate each GUI's widget with the information about who can execute its events taking also into account the organization of the GUI, in section 5.4 we derive a refined version of the two functions that automatically carry out such task, i.e., $AllowedRoles_G$ and $Constraint_G$. In section 5.5, we show how these functions can be constructed in OCL. In section 5.6, we use these functions to specify the smartsec transformation in QVT that automatically builds a smart, security-aware GUI model from a security model and the GUI model. Finally, in section 5.7, we talk about the correctness of this smartness transformation.
Figure 5.1 depicts the smartness transformation we discuss further on in more detail.

5.2 Smartness: Informal description

In this section, we provide our notion of smart, security-aware GUI designs. Moreover, we show how to transform a GUI into one that is smart and security-aware. Intuitively,

[1]Note that we consider here the security data model in abstract syntax including all the information explicitly.

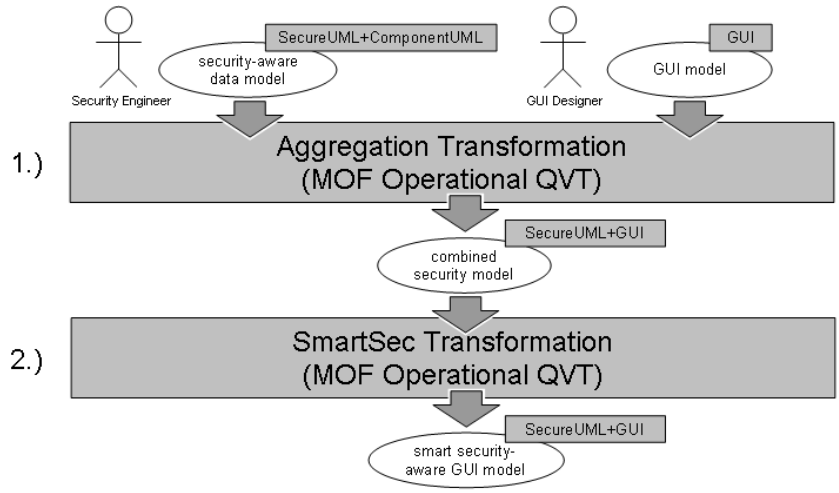

Figure 5.1: Generating smart, security-aware application GUIs.

a smart, security-aware GUI design is one where widgets' events are *annotated* with information about who can execute them based on the actions that these events trigger. In the case of events triggering application-data actions, the annotations should be *aware* of the application-data security policy. In the case of events triggering widget-actions, the annotations should be *smart*, that is, they should take into account the annotations attached to the events of the widgets which will be opened by the widget actions. We introduce a distinct kind of event, namely the *onCreate* event and some further assumptions in order to be able to reflect these situations in our modeling language. In section 5.3 this will be discussed precisely.

Figure 5.2 shows the same simple example of a GUI that we used in the previous chapter. This example GUI consists of two windows. The first window includes several buttons, button *Add employee* has been linked to an *onClick* event that fires an *open* widget action opening the second window. The second window includes two text entries and two buttons. Button *New* has been linked to an *onClick* event that fires several model actions. The security-policy defined by the security engineer defines the set of roles R_i that is allowed to execute the corresponding model actions on the data model. We depict the SmartSec transformation in figure 5.3. The crucial difference to what is obtained by the security transformation is made by the fact that now widget's events are taken into account as well. Intuitively we can say that only the set of roles that is allowed to perform every data model action fired by the *New* button should be allowed to execute the *AtomicExecute* action on the event that opens the second window. Therefore the permissions should be propagated through the control-flow of the GUI model.

5.3 Smartness: Formal definition

Recall that we already provided a formalization for GUIs in section 3.4.
Figure 5.4 shows an example window in detail visualizing the meaning of the *onCreate* events. We are ready to formally define smart, security aware GUI designs.
In what follows, *non-circular* GUI designs are those for which there are no cycles in the widget opening actions: that is, by design, no widget *wd* can be opened by an

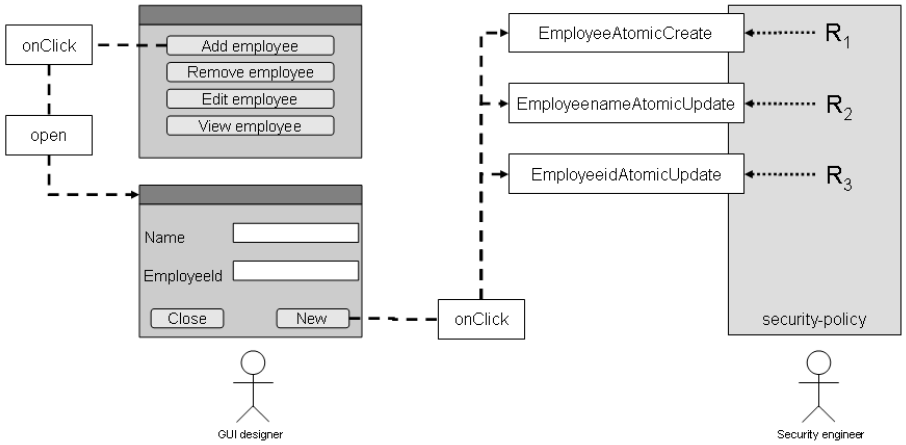

Figure 5.2: Example GUI before SmartSec Transformation.

action triggered by an event of a widget wd' which was opened by an action triggered by a sequence of events starting from an event of wd.

Definition 3 (Smart, Security-Awareness)**.** *Let $\mathcal{A}$ be an application protected by an access control policy. Then, a non-circular GUI design $\mathcal{G}$ for $\mathcal{A}$ is* smart, security-aware *if and only if there exist a function $EvAu_{\mathcal{G}}$ and a function $AuthConst_{\mathcal{G}}$ such that the following properties hold:*

i. For every widget wd, and event $ev \in Ev(wd)$ with $ev \not\equiv \text{onCreate}_{wd}$,

$$EvAu_{\mathcal{G}}(ev) = SecAware(ev) \cap Smart(ev),$$

where the sets of roles $SecAware(ev)$ and $Smart(ev)$ are defined as follows:

- *If $DaAc(ev) = \emptyset$. Then*

$$SecAware(ev) = \text{All}.$$

- *If $DaAc(ev) = \{daac_1, \ldots, daac_n\}$. Then*

$$SecAware(ev) = \bigcap_{i=1}^{n} DaAu_{\mathcal{A}}(daac_i).$$

- *If $WdAc(ev) = \emptyset$ or $WdAc(ev) = \bigcup_{i=1}^{n}\{(\text{close}, wd'_i)\}$. Then*

$$Smart(ev) = \text{All}.$$

- *If $WdAc(ev) = \bigcup_{i=1}^{j}\{(\text{open}, wd'_i)\} \cup \bigcup_{i=j+1}^{m}\{(\text{close}, wd'_i)\}$ where $j \geq 1$, let for $1 \leq i \leq j$, $CompEv(wd'_i) = \{ev'_{i_1}, \ldots, ev'_{i_n}\}$. Then*

$$Smart(ev) = \bigcap_{i=1}^{j} \left(\bigcap_{h=i_1}^{i_n} EvAu_{\mathcal{G}}(ev'_h) \right).$$

ii. For every widget wd, and event $ev \in Ev(wd)$ with $ev \equiv \text{onCreate}_{wd}$,

- *If wd is a non-container widget, with $Ev(wd) = \{\text{onCreate}_{wd}\}$ then*

$$EvAu_{\mathcal{G}}(ev) := \text{All}.$$

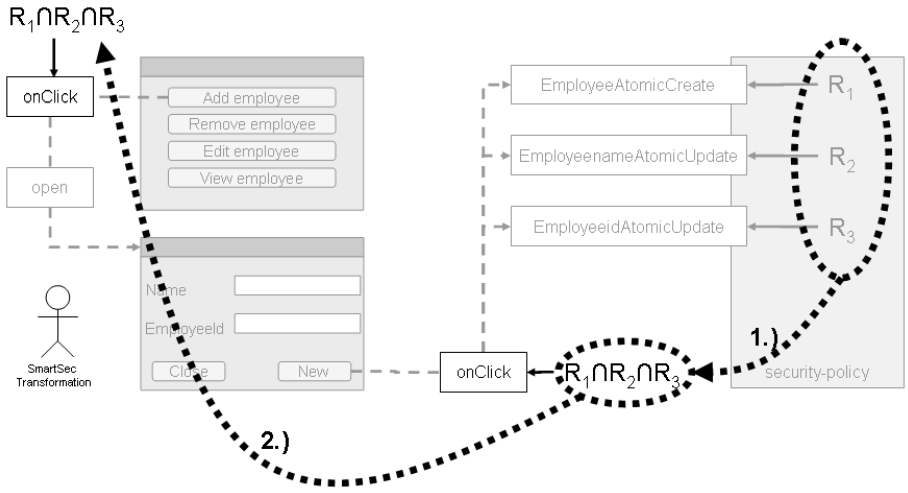

Figure 5.3: Example GUI during SmartSec Transformation.

- *If wd is a non-container widget, with* $Ev(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$*, and* $CompEv(wd) = \{onCreate\}$*, then*

$$EvAu_{\mathcal{G}}(ev) := \bigcup_{i=1}^{n} EvAu_{\mathcal{G}}(ev_i)\,.$$

- *If wd is a non-container widget, with* $CompEv(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$*, then*

$$EvAu_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} EvAu_{\mathcal{G}}(ev_i)\,.$$

- *If wd is a container widget, with* $In = \{wd_1, \ldots, wd_n\}$*, then,*

$$EvAu_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})\,.$$

iii. For each $rl \in EvAu_{\mathcal{G}}(ev)$ *with* $ev \not\equiv \text{onCreate}_{wd}$*,*

$$AuthConst_{\mathcal{G}}(rl, ev) = AuthSec(rl, ev) \wedge AuthSmart(rl, ev),$$

where the authorization constraints $AuthSec(rl, ev)$ *and* $AuthSmart(rl, ev)$ *are defined as follows:*

- *If* $DaAc(ev) = \emptyset$*. Then*

$$AuthSec(rl, ev) = \text{True}.$$

- *If* $DaAc(ev) = \{daac_1, \ldots, daac_n\}$*. Then*

$$AuthSec(rl, ev) = \bigwedge_{i=1}^{n} AuthConst_{\mathcal{A}}(rl, daac_i).$$

- *If* $WdAc(ev) = \emptyset$ *or* $WdAc(ev) = \bigcup_{i=1}^{n}\{(\text{close}, wd'_i)\}$*. Then*

$$AuthSmart(rl, ev) = \text{True}.$$

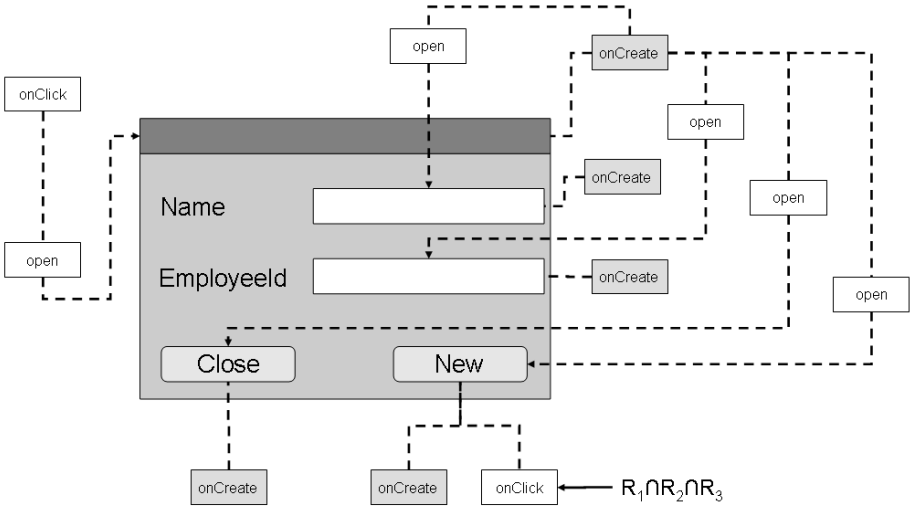

Figure 5.4: A simplified example window.

- *If* $WdAc(ev) = \bigcup_{i=1}^{j}\{(\text{open}, wd'_i)\} \cup \bigcup_{i=j+1}^{m}\{(\text{close}, wd'_i)\}$ *where* $j \geq 1$, *let for* $1 \leq i \leq j$, $CompEv(wd'_i) = \{ev'_{i_1}, \ldots, ev'_{i_n}\}$. *Then*

$$AuthSmart(rl, ev) = \bigwedge_{i=1}^{j}\left(\bigwedge_{h=i_1}^{i_n} AuthConst_{\mathcal{G}}(rl, ev'_h)\right).$$

iv. And for each $rl \in EvAu_{\mathcal{G}}(ev)$ *with* $ev \equiv \text{onCreate}_{wd}$,

- *If wd is a non-container widget, with* $Ev(wd) = \{\text{onCreate}_{wd}\}$ *then*

$$AuthConst_{\mathcal{G}}(rl, ev) := \text{True}.$$

- *If wd is a non-container widget, with* $Ev(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$, *and* $CompEv(wd) = \{onCreate\}$, *then*

$$AuthConst_{\mathcal{G}}(rl, ev) := \bigvee_{i=1}^{n} AuthConst_{\mathcal{G}}(rl, ev_i).$$

- *If wd is a non-container widget, with* $CompEv(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$, *then*

$$AuthConst_{\mathcal{G}}(rl, ev) := \bigwedge_{i=1}^{n} AuthConst_{\mathcal{G}}(rl, ev_i).$$

- *If wd is a container widget, with* $In = \{wd_1, \ldots, wd_n\}$, *then,*

$$AuthConst_{\mathcal{G}}(rl, ev) := \bigwedge_{i=1}^{n} AuthConst_{\mathcal{G}}(rl, \text{onCreate}_{wd_i}).$$

5.4 New construction of $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$

Starting from a SecureUML+ComponentUML data model and a GUI model, again one cannot expect the GUI designer to annotate each widget with the information about who can execute its events: first of all, because he may not be fully aware of the application data security policy, but also because this is a cumbersome and prone to error task, if done manually.
In this section we define two functions to carry out automatically such task, we call these functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$. The former function is aimed to compute automatically for each GUI event which roles can execute it according to the permissions established on the data actions and widget actions associated to the given event in the data model. The latter function is aimed to compute automatically for each role with granted access to execute an event under which constraints it is actually allowed to execute it. These functions can be seen as a refinement of the ones defined in section 4.4 by taking into account the widget actions. Of course, our aim with such definitions is that the above defined properties for smart, security-awareness hold.

Remark 3. *Let $\mathcal{A}$ be an application protected by an access control policy. Then, given a non-circular GUI design $\mathcal{G}$ for $\mathcal{A}$ we can always construct a function $AllowedRoles_{\mathcal{G}}()$ that can be used to make $\mathcal{G}$ smart, security-aware.*

The construction is as follows:

- For every event ev, $ev \not\equiv$ onCreate,
  - If $DaAc(ev) = \emptyset$ and either $WdAc(ev) = \emptyset$ or $WdAc(ev) = \bigcup_{i=1}^{m}\{(\text{close}, wd_i)\}$, then
  $$AllowedRoles_{\mathcal{G}}(ev) := \text{All}\,.$$
  - If $DaAc(ev) = \{daac_1, \ldots, daac_n\}$ and either $WdAc(ev) = \emptyset$ or $WdAc(ev) = \bigcup_{i=1}^{m}\{(\text{close}, wd_i)\}$, then
  $$AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} DaAu_{\mathcal{A}}(daac_i)\,.$$
  - If $DaAc(ev) = \emptyset$ and $WdAc(ev) = \bigcup_{i=1}^{j}\{(\text{open}, wd_i)\} \cup \bigcup_{i=j+1}^{m}\{(\text{close}, wd_i)\}$ where $j \geq 1$, then
  $$AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{j} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i})\,.$$
  - If $DaAc(ev) = \{daac_1, \ldots, daac_n\}$ and $WdAc(ev) = \bigcup_{i=1}^{j}\{(\text{open}, wd_i)\} \cup \bigcup_{i=j+1}^{m}\{(\text{close}, wd_i)\}$, where $j \geq 1$, then
  $$AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} DaAu_{\mathcal{A}}(daac_i) \cap \bigcap_{i=1}^{j} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i})\,.$$
- For every event ev, $ev \equiv \text{onCreate}_{wd}$,
  - If wd is a non-container widget, with $Ev(wd) = \{\text{onCreate}_{wd}\}$ then
  $$AllowedRoles_{\mathcal{G}}(ev) := \text{All}.$$

- If wd is a non-container widget, with $Ev(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$, and $CompEv(wd) = \{onCreate\}$, then

$$AllowedRoles_{\mathcal{G}}(ev) := \bigcup_{i=1}^{n} AllowedRoles_{\mathcal{G}}(ev_i)\,.$$

- If wd is a non-container widget, with $CompEv(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$, then

$$AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} AllowedRoles_{\mathcal{G}}(ev_i)\,.$$

- If wd is a container widget, with $In = \{wd_1, \ldots, wd_n\}$, then,

$$AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i})\,.$$

Remark 4. *Let $\mathcal{A}$ be an application. Then, given a non-circular GUI design $\mathcal{G}$ for $\mathcal{A}$ we can always construct a function $Constraint_{\mathcal{G}}()$ that can be used to make $\mathcal{G}$ smart, security-aware.*

The construction is as follows:

- For every event ev, $ev \not\equiv \text{onCreate}$,
  - If $DaAc(ev) = \emptyset$ and either $WdAc(ev) = \emptyset$ or $WdAc(ev) = \bigcup_{i=1}^{m}\{(\text{close}, wd_i)\}$, then

$$Constraint_{\mathcal{G}}(rl, ev) := \text{True}\,.$$

  - If $DaAc(ev) = \{daac_1, \ldots, daac_n\}$ and either $WdAc(ev) = \emptyset$ or $WdAc(ev) = \bigcup_{i=1}^{m}\{(\text{close}, wd_i)\}$, then

$$Constraint_{\mathcal{G}}(rl, ev) := \bigwedge_{i=1}^{n} AuthConst_{\mathcal{A}}(rl, daac_i)\,.$$

  - If $DaAc(ev) = \emptyset$ and $WdAc(ev) = \bigcup_{i=1}^{j}\{(\text{open}, wd_i)\} \cup \bigcup_{i=j+1}^{m}\{(\text{close}, wd_i)\}$ where $j \geq 1$, then

$$Constraint_{\mathcal{G}}(rl, ev) := \bigwedge_{i=1}^{j} Constraint_{\mathcal{G}}(rl, \text{onCreate}_{wd_i})\,.$$

  - If $DaAc(ev) = \{daac_1, \ldots, daac_n\}$ and $WdAc(ev) = \bigcup_{i=1}^{j}\{(\text{open}, wd_i)\} \cup \bigcup_{i=j+1}^{m}\{(\text{close}, wd_i)\}$ where $j \geq 1$, then

$$Constraint_{\mathcal{G}}(rl, ev) := \bigwedge_{i=1}^{n} AuthConst_{\mathcal{A}}(rl, daac_i) \wedge \bigwedge_{i=1}^{j} Constraint_{\mathcal{G}}(rl, \text{onCreate}_{wd_i})\,.$$

- For every event ev, $ev \equiv \text{onCreate}_{wd}$,
  - If wd is a non-container widget, with $Ev(wd) = \{\text{onCreate}_{wd}\}$ then

$$Constraint_{\mathcal{G}}(rl, ev) := \text{True}.$$

– If wd is a non-container widget, with $Ev(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$, and $CompEv(wd) = \{onCreate\}$, then

$$Constraint_{\mathcal{G}}(rl, ev) := \bigvee_{i=1}^{n} Constraint_{\mathcal{G}}(rl, ev_i)\,.$$

– If wd is a non-container widget, with $CompEv(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$, then

$$Constraint_{\mathcal{G}}(rl, ev) := \bigwedge_{i=1}^{n} Constraint_{\mathcal{G}}(rl, ev_i)\,.$$

– If wd is a container widget, with $In = \{wd_1, \ldots, wd_n\}$, then,

$$Constraint_{\mathcal{G}}(rl, ev) := \bigwedge_{i=1}^{n} Constraint_{\mathcal{G}}(rl, \text{onCreate}_{wd_i})\,.$$

5.5 Construction of the functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$ in OCL

In chapter 3 we have already introduced our security GUI modeling language, i.e. SecureUML+GUI together with a list of operations for analyzing its models using OCL. In what follows we will again refer to these definitions in order to implement the above formally described functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$ in OCL. As our model transformation later on is defined using operational QVT and OCL is part of QVT, we will use this implementation to transform the GUI model into a smart, security-aware GUI model.

Implementation of function $AllowedRoles_{\mathcal{G}}$. The following queries implement the refined function $AllowedRoles_{\mathcal{G}}$ using OCL:

```
context Event::AllowedRolesG():Set(Role)
body:
if not self.event = EventEnum::onCreate then
  if self.DaAc()->isEmpty() and self.WdAc()->
    select(a | a.guiAction = GUIAction::open)->isEmpty() then
    Role.allInstances()
  else
    if not self.DaAc()->isEmpty() and self.WdAc()->
      select(a | a.guiAction = GUIAction::open)->isEmpty() then
      setIntersection(
        self.DaAc().modelAction->collectNested(daac | daac.DaAu())
        ->asSet()
      )
    else
      if self.DaAc()->isEmpty() and not self.WdAc()->
        select(a | a.guiAction = GUIAction::open)->isEmpty() then
        setIntersection(
          self.WdAc()->select(a | a.guiAction =
            GUIAction::open).actionOn->collectNested(wd | wd.Ev()->
            select(e | e.event = EventEnum::onCreate)->
            asOrderedSet()->first().AllowedRolesG())->asSet()
        )
```

```
      else
        setIntersection(
          self.DaAc().modelAction->collectNested(daac | daac.DaAu())
          ->asSet()
        )->intersection(
          setIntersection(
            self.WdAc()->select(a | a.guiAction =
              GUIAction::open).actionOn->collectNested(wd | wd.Ev()->
              select(e | e.event = EventEnum::onCreate)->
            asOrderedSet()->first().AllowedRolesG())->asSet()
          )
        )
      endif
    endif
  endif
else
  if not self.holder.oclIsKindOf(Container) and
    self.holder.Ev()->excluding(self.holder.Ev()->
    select(e | e.event = EventEnum::onCreate)->
           asOrderedSet()->first())->isEmpty() then
    Role.allInstances()
  else
    if not self.holder. oclIsKindOf(Container) and
      not self.holder.Ev()->excluding(self.holder.Ev()->
      select(e | e.event = EventEnum::onCreate)->
           asOrderedSet()->first())->isEmpty() and
      self.holder.CompEv()->excluding(self.holder.Ev()->
      select(e | e.event = EventEnum::onCreate)->
           asOrderedSet()->first())->isEmpty() then
      self.holder.Ev()->excluding(self.holder.Ev()->
        select(e | e.event = EventEnum::onCreate)->
           asOrderedSet()->first())->
        collect(e | e.AllowedRolesG())->asSet()
    else
      if not self.holder. oclIsKindOf(Container) and
        not self.holder.CompEv()->excluding(self.holder.Ev()->
        select(e | e.event = EventEnum::onCreate)->
           asOrderedSet()->first())->isEmpty() then
        setIntersection(
          self.holder.CompEv()->excluding(self.holder.Ev()->
            select(e | e.event = EventEnum::onCreate)->
            asOrderedSet()->first())->
            collectNested(ev | ev.AllowedRolesG())->asSet()
        )
      else
        if self.holder. oclIsKindOf(Container) then
          setIntersection(
            self.holder.oclAsType(Container).In()->collect(wd | wd.Ev()->
              select(e | e.event = EventEnum::onCreate))->
              collectNested(ev | ev.AllowedRolesG())->asSet()
          )
        endif
      endif
    endif
```

```
  endif
endif

context setIntersection(rs : Set(Set(Role))):Set(Role)
body: if rs->size() = 1 then rs->asOrderedSet()->first()
  else rs->asOrderedSet()->first()->intersection(
    setIntersection(rs->excluding(rs->asOrderedSet()->first()))
  ) endif
```

Implementation of function $Constraint_{\mathcal{G}}$**.** Analogous to the above definition we can define the function $Constraint_{\mathcal{G}}$ in OCL.

```
context Event::ConstraintG(rl:Role):String
body:
if not self.event = EventEnum::onCreate then
  if self.DaAc()->isEmpty() and self.WdAc()->
    select(a | a.guiAction = GUIAction::open)->isEmpty() then
    'True'
  else
    if not self.DaAc()->isEmpty() and self.WdAc()->
      select(a | a.guiAction = GUIAction::open)->isEmpty() then
      setConjunction(
        self.DaAc().modelAction->collect(daac | daac.AuthConstA(rl))
        ->asSet()
      )
    else
      if self.DaAc()->isEmpty() and not self.WdAc()->
        select(a | a.guiAction = GUIAction::open)->isEmpty() then
        setConjunction(
          self.WdAc()->select(a | a.guiAction =
            GUIAction::open).actionOn->collect(wd | wd.Ev()->
            select(e | e.event = EventEnum::onCreate)->asOrderedSet()->
            first().ConstraintG(rl))->asSet()
        )
      else
        '('.concat(
          setConjunction(
            self.DaAc().modelAction->collect(daac | daac.AuthConstA(rl))
            ->asSet()
          )).concat(')␣and␣('.concat(
          setConjunction(
            self.WdAc()->select(a | a.guiAction =
              GUIAction::open).actionOn->collect(wd | wd.Ev()->
              select(e | e.event = EventEnum::onCreate)->asOrderedSet()->
              first().ConstraintG(rl))->asSet()
          ).concat(')')
        )
      endif
    endif
  endif
else
  if not self.holder. oclIsKindOf(Container) and
    self.holder.Ev()->excluding(self.holder.Ev()->
    select(e | e.event = EventEnum::onCreate)->
```

```
            asOrderedSet()->first())->isEmpty() then
    'True'
  else
    if not self.holder. oclIsKindOf(Container) and
      not self.holder.Ev()->excluding(self.holder.Ev()->
      select(e | e.event = EventEnum::onCreate)->
            asOrderedSet()->first())->isEmpty() and
      self.holder.CompEv()->excluding(self.holder.Ev()->
      select(e | e.event = EventEnum::onCreate)->
            asOrderedSet()->first())->isEmpty() then
      setDisjunction(
        self.holder.Ev()->excluding(self.holder.Ev()->
          select(e | e.event = EventEnum::onCreate)->
            asOrderedSet()->first())->
          collect(e | e.ConstraintG(rl))->asSet()
      )
    else
      if not self.holder. oclIsKindOf(Container) and
        not self.holder.CompEv()->excluding(self.holder.Ev()->
        select(e | e.event = EventEnum::onCreate)->
            asOrderedSet()->first())->isEmpty() then
        setConjunction(
          self.holder.CompEv()->excluding(self.holder.Ev()->
            select(e | e.event = EventEnum::onCreate)->
            asOrderedSet()->first())->
            collect(ev | ev.ConstraintG(rl))->asSet()
        )
      else
        if self.holder. oclIsKindOf(Container) then
          setConjunction(
            self.holder.oclAsType(Container).In()->collect(wd | wd.Ev()->
              select(e | e.event = EventEnum::onCreate))->
              collect(ev | ev.ConstraintG(rl))->asSet()
          )
        endif
      endif
    endif
  endif
endif
```

Table 5.1 shows the correspondence between the formal definitions and the evaluation of OCL expressions on instances of the combined metamodel [2]

5.6 Smartness transformation

In this section we describe the model transformation that automatically generates smart, security-aware SecureUML+GUI models from SecureUML+ComponentUML

[2]Rigorously proving this correspondence requires detailed meta-reasoning that involves both the semantics of the underlying formal system, the semantics of OCL, and the translation scheme from terms in the semantic domain to OCL expressions. This is a large undertaking and outside the scope of this thesis. In many practical cases however, one may settle for the next best thing: it may be sufficient to have a careful understanding of the metamodel of the modeling languages.

it is satisfied	it evaluates to **true**
$rl \in AllowedRoles_{\mathcal{G}}(ev)$	ev.`AllowedRolesG()->includes(`rl`)`
$ac = Constraint_{\mathcal{G}}(rl, ev)$	ev.`ConstraintG(`rl`)` $= ac$

Table 5.1: Mapping to OCL functions

models and GUI models. We assume here, as mentioned above, that the source models have the same ComponentUML application-data model. Next, we implement in operational QVT [15, Section 8.4.6] the model transformation that we have illustrated in the previous sections. This automated transformation carries out the last step of the high-level process description given in the introduction of this chapter. For simplicity we split again the transformation into two sequential transformations:

1. Aggregation of the GUI model and the SecureUML+ComponentUML model into a single combined security model.
2. Permission lifting from the SecureUML+ComponentUML part of the model to the SecureUML+GUI part.

The first transformation takes as source models the SecureUML+ComponentUML model defined by the security engineer and the GUI model defined by the GUI designer and simply aggregates the information of these two models into one combined security model (see section 3.3.2). Figure 5.1 shows an excerpt of this QVT transformation. Then, in the second transformation, we add to the resulting combined security model all the permissions that are required in order to satisfy the smart, security-aware properties formally defined in the section 5.3. Here, to decide whether a *Role* should be linked to a *Permission* granting access to *AtomicExecute* an *Event*, we use again QVT functions that, basically, implement the construction of the functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$ defined in section 5.3 as shown in the previous section. Here, we only describe the implementation. The full QVT transformation with all its mapping functions, can be found in Appendix C.

Listing 5.1: The Aggregation Transformation in QVTO syntax (Excerpt).

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

transformation AggregationTransformation(in guiModel : GUI,
  in secModel : SECUMLANDCOMPUML, out guiSecModel : SECUMLANDGUI);

main() {
  secModel.objects()[SECUMLANDCOMPUML::User]->map User_to_User();
  secModel.objects()[SECUMLANDCOMPUML::Role]->map Role_to_Role();
  ...
}
mapping SECUMLANDCOMPUML::User::User_to_User() : SECUMLANDGUI::User
{
  name := self.name;
}

mapping SECUMLANDCOMPUML::Role::Role_to_Role() : SECUMLANDGUI::Role
{
  name := self.name;
  default := self.default;
```

```
  includes := self.includes.resolve()->oclAsType(SECUMLANDGUI::User)->
    asOrderedSet();
}
...
}
```

In Figure 5.2 we present the heading of the smartness transformation that we describe here together with the mapping functions that are explained now in more detail. After the aggregation step whose heading is depicted in figure 5.1, we have all the information simply integrated into one model that gives us the possibility to analyze the resulting model like it is proposed in [1]. The metamodel `SECUMLANDGUI` is the SecureUML+GUI metamodel which was introduced in chapter 3. Again, the operational transformation is defined by mapping functions, which are executed sequentially. We only describe these functions here, their full definitions are also available in Appendix C. The final target model is obtained in the following two steps:

Step 1: Aggregation Transformation.[3]

- Every element in the (source) SecureUML+ComponentUML model are copied, along with their hierarchies, into the (target) combined security model, using the corresponding mapping functions of the form.

  ```
  secModel.objects()[SECUMLANDCOMPUML::User]->map User_to_User();
  secModel.objects()[SECUMLANDCOMPUML::Role]->map Role_to_Role();
  ```

  ...
- Every element in the (source) GUI model are copied, along with their hierarchies, into the (target) combined security model, using the corresponding mapping functions of the same form as above.
- The links between the*ModelActions* of the GUI model part and the concrete data model *AtomicActions* of the SecureUML+GUI model part are being established.

Step 2: SmartSec Transformation.

- For each *Event* in the SecureUML+GUI model, and for each *Role* that is allowed to perform the *AtomicExecute* action assigned to the *Event* according to the function $AllowedRoles_{\mathcal{G}}()$[4], a *Permission* is created in the SecureUML+GUI model that grants access to the *Role* to *AtomicExecute* the *Event*. The *AuthorizationConstraint* constraining the newly created *Permission* is then being computed using the function $Constraint_{\mathcal{G}}()$. This is accomplished using the following mapping functions:

  ```
  smartSecGUI.objects()[Event]->map liftPermissions();
  ```

  Where `liftPermissions()` is defined as follows:

  ```
  mapping Event::liftPermissions() : Set(Permission)
  {
    init{
      result := self.AllowedRolesG()->setPermission(self)->asSet();
    }
  }
  ```

[3]Note that we defined QVT queries to check that the outcome of the aggregation transformation is correct according to the definitions of a secure GUI modeling language above. The definitions of these queries can be found in Appendix C.

[4]The function $AllowedRoles_{\mathcal{G}}()$ has been implemented according to the construction section that follows below. The full implementation using QVTO can be seen in Appendix C.

Listing 5.2: The SmartSec Transformation's heading in QVTO syntax.

```
modeltype SECUMLANDGUI uses "http://secumlandgui/1.0";

transformation SmartSecTransformation(inout smartSecGUI : SECUMLANDGUI);

main() {
  smartSecGUI.objects()[Event]->map liftPermissions();
}

mapping Event::liftPermissions() : Set(Permission)
{
  init{
    result := self.allowedRolesG()->setPermission(self)->asSet();
  }
}

mapping Role::setPermission(e: Event) : Permission
{
  name := self.name + e.holder.label + e.event.repr() + 'Execute';
  accesses := e.actions;
  default := false;
  givesAccess := self;
  isConstraintBy := self.map setAuthConst(e);
}

mapping Role::setAuthConst(e: Event) : AuthorizationConstraint
{
  name := self.name + e.holder.label + e.event.repr() + 'ExecutionConstraint';
  body := e.ConstraintG(self);
  language := 'some_language';
}
}
```

So far the transformation seems the same that we presented in the last chapter. The difference lays in the refined functions $AllowedRoles_{\mathcal{G}}()$ and $Constraint_{\mathcal{G}}()$. These functions should now return the proper access-control information according to the definition of smart, security-awareness we defined in the section 5.3. Again, the resulting SecureUML+GUI model can then be used for either code generation or to analyze the validity of the GUI model that has been designed. In [1] it is explained in detail how SecureUML models can be automatically analyzed using OCL queries, some other interesting analysis operations were provided in section 3.3.5.

5.7 Correctness

We claim that our smartness transformation is correct, i.e. the GUI model obtained as an output of the transformation is smart, security-aware w.r.t. definition 3 under the assumption of the correspondences between the formal functions and the corresponding OCL operations.

Definition 4 (Chain of containers)**.** *Given* $\{wd_1, wd_2, \ldots, wd_n\}$ *a set of container widgets, we say that they are a chain of containers if and only if* $wd_{i+1} \in In(wd_i)$ *for all* i *with* $1 \leq i \leq n-1$, $n \geq 2$ *where* wd_n *is a container such that* $In(wd_n)$ *is a set of non container widgets.*

We define the length of a chain of containers $length(\{wd_1, wd_2, \ldots, wd_n\}) = n - 1$. *Given a set of chains of containers* $\{c_1, c_2, \ldots, c_k\}$, *we define its depth as* $depth(\{c_1, c_2, \ldots, c_k\}) = max_{h=1}^{k}\{length(c_h)\}$.

Theorem 2 (Correctness)**.** *Let* $\mathcal{A}$ *be an application protected by an access control policy. Let* $\mathcal{G}$ *be a GUI design for* $\mathcal{A}$ *obtained as the output model of the smartness transformation. Then,* $\mathcal{G}$ *is smart, security-aware according to definition 3.*

Proof. We build the functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$ defined in remarks 3 and 4 for the GUI design $\mathcal{G}$. We have to prove that this functions make $\mathcal{G}$ smart and security-aware according to definition 3, i.e.,

1. For every event ev, $ev \not\equiv$ onCreate, we know that $EvAu_{\mathcal{G}}(ev) = SecAware(ev) \cap Smart(ev)$ by definition 3. We want to prove that

$$AllowedRoles_{\mathcal{G}}(ev) = EvAu_{\mathcal{G}}(ev) \tag{5.1}$$

   We make the proof by case distinction:

   (a) If $DaAc(ev) = \emptyset$ and either $WdAc(ev) = \emptyset$ or $WdAc(ev) = \bigcup_{i=1}^{m}\{(\text{close}, wd_i)\}$. By remark 3, we know that $AllowedRoles_{\mathcal{G}}(ev) =$ All and, by definition 3, we know also that $SecAware(ev) =$ All and $Smart(ev) =$ All. Therefore (5.1) holds.

   (b) If $DaAc(ev) = \{daac_1, \ldots, daac_n\}$ and either $WdAc(ev) = \emptyset$ or $WdAc(ev) = \bigcup_{i=1}^{m}\{(\text{close}, wd_i)\}$.
   By remark 3, we know that $AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} DaAu_{\mathcal{A}}(daac_i)$ and, by definition 3, we know also that $SecAware(ev) = \bigcap_{i=1}^{n} DaAu_{\mathcal{A}}(daac_i)$ and $Smart(ev) =$ All. Therefore (5.1) holds.

   (c) If $DaAc(ev) = \emptyset$ and $WdAc(ev) = \bigcup_{i=1}^{j}\{(\text{open}, wd_i)\} \cup \bigcup_{i=j+1}^{m}\{(\text{close}, wd_i)\}$ where $j \geq 1$.
   In this case what we know by remark 3 is that $AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{j} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i})$ and by definition 3 we know also that $SecAware(ev) =$ All and $Smart(ev) = \bigcap_{i=1}^{j}\left(\bigcap_{h=i_1}^{i_n} EvAu_{\mathcal{G}}(ev'_h)\right)$ where $CompEv(wd_i) = \{ev_{i_1}, \ldots, ev_{i_n}\}$. Thus, it is enough to show that for all i with $1 \leq i \leq j$

$$AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) = \bigcap_{h=i_1}^{i_n} EvAu_{\mathcal{G}}(ev_h) \tag{5.2}$$

   $ev \equiv \text{onCreate}_{wd_i}$

   - If wd_i is a non-container widget, with $Ev(wd_i) = \{\text{onCreate}_{wd_i}\}$. Since $CompEv(wd_i) = \{onCreate_{wd_i}\}$, $\bigcap_{h=i_1}^{i_n} EvAu_{\mathcal{G}}(ev_h) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$. By case 2a) we know that $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$. Therefore the equation (5.2) holds.
   - If wd_i is a non-container widget, with $Ev(wd_i) \setminus \{\text{onCreate}_{wd_i}\} = \{ev_1, \ldots, ev_n\}$, and $CompEv(wd_i) = \{onCreate_{wd_i}\}$.
   Since $CompEv(wd_i) = \{onCreate_{wd_i}\}$, $\bigcap_{h=i_1}^{i_n} EvAu_{\mathcal{G}}(ev_h) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$.
   By case 2b) we know that $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$. Therefore the equation (5.2) holds.

- If wd_i is a non-container widget, with $CompEv(wd_i)\backslash\{\text{onCreate}_{wd_i}\} = \{ev_1, \ldots, ev_n\}$.
  $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) = \bigcap_{h=1}^{n} AllowedRoles_{\mathcal{G}}(ev_h)$. Since for all h $ev_h \not\equiv \text{onCreate}_{wd_i}$ and wd_i is a non-container widget, we know that $AllowedRoles_{\mathcal{G}}(ev_h) = EvAu_{\mathcal{G}}(ev_h)$ holds for all h by cases 1a) and 1b). Therefore the equation (5.2) holds.
- If wd_i is a container widget, with $In(wd_i) = \{wd_1^i, \ldots, wd_{n_i}^i\}$.
  $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) := \bigcap_{h=1}^{n} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_h^i})$ and $\bigcap_{h=i_1}^{i_n} EvAu_{\mathcal{G}}(ev_h) = \bigcap_{h=i_1}^{i_n} EvAu_{\mathcal{G}}(\text{onCreate}_{wd_h^i})$. Thus, it is enough to show that for all h

$$AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_h^i}) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_h^i}) \tag{5.3}$$

  - if wd_h^i is a non container widget the equation (5.3) holds by application of 2a), 2b) or 2c).
  - if wd_h^i is a is a container widget, the equation (5.3) holds by case (2d).

  Therefore (5.1) holds.

(d) If $DaAc(ev) = \{daac_1, \ldots, daac_n\}$ and $WdAc(ev) = \bigcup_{i=1}^{j}\{(\text{open}, wd_i)\} \cup \bigcup_{i=j+1}^{m}\{(\text{close}, wd_i)\}$, where $j \geq 1$.
We know by remark 3 is that $AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} DaAu_{\mathcal{A}}(daac_i) \cap \bigcap_{i=1}^{j} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i})$ and what we know as well by definition 3 is that $SecAware(ev) = \bigcap_{i=1}^{n} DaAu_{\mathcal{A}}(daac_i)$ and $Smart(ev) = \bigcap_{i=1}^{j}\left(\bigcap_{h=i_1}^{i_n} EvAu_{\mathcal{G}}(ev'_h)\right)$. Due to the identity between the first two members of the intersection and by the previous case 1c), we can say that (5.1) holds.

2. For every event ev, $ev \equiv \text{onCreate}$,

(a) If wd is a non-container widget, with $Ev(wd) = \{\text{onCreate}_{wd}\}$.
By remark 3 we know that $AllowedRoles_{\mathcal{G}}(ev) = \text{All}$ and by definition 3 we know also that $EvAu_{\mathcal{G}}(ev) = \text{All}$. Therefore (5.1) holds.

(b) If wd is a non-container widget, with $Ev(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$, and $CompEv(wd) = \{onCreate\}$.
By remark 3 we know that $AllowedRoles_{\mathcal{G}}(ev) := \bigcup_{i=1}^{n} AllowedRoles_{\mathcal{G}}(ev_i)$ and by definition 3 we know also that $EvAu_{\mathcal{G}}(ev) := \bigcup_{i=1}^{n} EvAu_{\mathcal{G}}(ev_i)$.

- Then, it is enough to prove that for all $1 \leq i \leq n$ $AllowedRoles_{\mathcal{G}}(ev_i) = EvAu_{\mathcal{G}}(ev_i)$. Since wd is a non-container widget, we know that $ev_i \not\equiv \text{onCreate}_{wd}$ and either $WdAc(ev_i) = \emptyset$ or $WdAc(ev_i) = \bigcup_{j=1}^{k}\{(\text{close}, wd_j)\}$. Therefore the equation holds by case 1a) or 1b) for all i.

Therefore (5.1) holds.

(c) If wd is a non-container widget, with $CompEv(wd) \setminus \{\text{onCreate}_{wd}\} = \{ev_1, \ldots, ev_n\}$.
By remark 3 we know that $AllowedRoles_{\mathcal{G}}(ev) = \bigcap_{i=1}^{n} AllowedRoles_{\mathcal{G}}(ev_i)$ and by definition 3 we know also that $EvAu_{\mathcal{G}}(ev) = \bigcap_{i=1}^{n} EvAu_{\mathcal{G}}(ev_i)$.

- Again, it is enough to prove that for all $1 \leq i \leq n$ $AllowedRoles_{\mathcal{G}}(ev_i) = EvAu_{\mathcal{G}}(ev_i)$. Since wd is a non-container widget, we know that $ev_i \not\equiv$

onCreate$_{wd}$ and either $WdAc(ev_i) = \emptyset$ or $WdAc(ev_i) = \bigcup_{j=1}^{k}\{(\text{close}, wd_j)\}$. Therefore the equation holds by case 1a) or 1b) for all i.

Therefore (5.1) holds.

(d) If wd is a container widget, with $In(wd) = \{wd_1, \ldots, wd_n\}$. We need to prove that

$$AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd}) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd}) \tag{5.4}$$

By remark 3 and definition 3 we know that $AllowedRoles_{\mathcal{G}}(ev) := \bigcap_{i=1}^{n} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i})$ and we know as well that $EvAu_{\mathcal{G}}(ev) = \bigcap_{i=1}^{n} EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$ (resp.). Under the assumption of non-circularity, let $Chains(wd) = \{c_1, \ldots, c_k\}$ be all the chains of containers starting from wd. Let $length(c_i) = l_i$; $\mathcal{S} = depth(Chains(wd)) = max_{i=i}^{k}\{l_i\}$. We are going to prove 5.4 by induction in $\mathcal{S}$:

i. Before starting the induction, consider that for each wd_i contained in $In(wd)$, wd_i is a non container widget. Then we know that for each i $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$ by application of 2a), 2b) or 2c). Since $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd}) = \bigcap_{i=1}^{n} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i})$ and we also know that $EvAu_{\mathcal{G}}(\text{onCreate}_{wd}) = \bigcap_{i=1}^{n} EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$, the equation (5.4) holds.

ii. Now, if $\mathcal{S} = 2$; we can assume that for all i with $j+1 \leq i \leq n$ and $n-(j+1) \geq 0$, wd_i is a non container widget, thus $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$ holds by case 2(d)i). For all i with $1 \leq i \leq j$ and $j \geq 1$, we consider the chains of the form $c_i = \{wd, wd_i\}$, where $In(wd_i) = \{wd_1^i, \ldots, wd_n^i\}$ and for all h, wd_h^i is a non-container widget since $\mathcal{S} = 2$. By case 2(d)i, we know that $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_h^i}) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_h^i})$ for all h. For all i with $1 \leq i \leq j$, by remark 3 and definition 3 we know that $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) = \bigcap_{h=1}^{n} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_h^i})$ and $EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i}) = \bigcap_{h=1}^{n} EvAu_{\mathcal{G}}(\text{onCreate}_{wd_h^i})$, then $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$ for all i. Therefore (5.4) holds.

iii. If $\mathcal{S} > 2$; we can assume that

- for all i with $j+m+1 \leq i \leq n$ and $n-(j+m+1) \geq 0$, wd_i is a non container widget, thus $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_i}) = EvAu_{\mathcal{G}}(\text{onCreate}_{wd_i})$ holds by case 2(d)i);
- for all i with $j \leq i \leq j+m$ and $m \geq 0$, we call $i = i_1$ and consider chain $c_h = \{wd, wd_{i_1}, wd_{i_2}^{i_1}, wd_{i_3}^{i_1 i_2}, \ldots, wd_{i_{l_h-1}}^{i_1 i_2 \ldots i_{l_h-2}}, wd_{i_{l_h}}^{i_1 i_2 \ldots i_{l_h-1}}\}$ with $l_h < \mathcal{S}$. Thus, by induction hypothesis, $EvAu_{\mathcal{G}}(\text{onCreate}_{wd_{i_1}}) = AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_{i_1}})$ (*) for all i_1.
- for all i with $1 \leq i \leq j-1$ and $j \geq 1$, we call $i = i_1$ and consider the chains whose length $l_h = \mathcal{S}$, that are of the form $\{wd, wd_{i_1}, wd_{i_2}^{i_1}, wd_{i_3}^{i_1 i_2}, \ldots, wd_{i_{l-1}}^{i_1 i_2 \ldots i_{l-2}}, wd_{i_{\mathcal{S}}}^{i_1 i_2 \ldots i_{\mathcal{S}-1}}\}$. We apply induction hypothesis to all the chains starting from wd_{i_1}, that are of the form $\{wd_{i_1}, wd_{i_2}^{i_1}, wd_{i_3}^{i_1 i_2}, \ldots, wd_{i_{l-1}}^{i_1 i_2 \ldots i_{l-2}}, wd_{i_{\mathcal{S}}}^{i_1 i_2 \ldots i_{\mathcal{S}-1}}\}$ whose length is $\mathcal{S}-1 < \mathcal{S}$. We obtain that $EvAu_{\mathcal{G}}(\text{onCreate}_{wd_{i_2}^{i_1}}) = AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_{i_2}^{i_1}})$ for all i_2. Since by remark 3 and

definition 3 $EvAu_{\mathcal{G}}(\text{onCreate}_{wd_{i_1}}) = \bigcap_{i_2} EvAu_{\mathcal{G}}(\text{onCreate}_{wd_{i_2}^{i_1}})$ and $AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_{i_1}}) = \bigcap_{i_2} AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_{i_2}^{i_1}})$, we obtain that $EvAu_{\mathcal{G}}(\text{onCreate}_{wd_{i_1}}) = AllowedRoles_{\mathcal{G}}(\text{onCreate}_{wd_{i_1}})$ for all i_1 in this case.

Therefore (5.4) holds.

Similar proving procedure is required to show that the function $Constraint_{\mathcal{G}}(rl, ev)$ defined in remark 4 is equal to the function $AuthConst_{\mathcal{G}}(rl, ev)$ provided in definition 3. □

Therefore the outcome of our transformation is smart, security-aware w.r.t. definition 3.

Again we implemented OCL queries to validate the outcome of the transformations against definition 3. The implementation of these QVTO queries can be seen in Appendix C. These queries can be used to check the correctness of the specific transformation output as a post-condition check.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Although creating user interfaces is a common task in application development and therefore 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, until now there has been no research into the systematic design of security aware GUIs. In this work, we have presented a contribution to this research area, i.e., to the research in the systematic design of GUIs whose functionality should adhere to the security policy designed for an underlying application-data model.
We have presented an approach based on model-transformation for automatically generating smart, security-aware GUI models. Indeed, we provide a formal definition of what does security-awareness and smartness mean in this setting. Based on these definitions, we proposed concrete model transformations that given a (source) security-aware data model and a (source) GUI model automatically generate a smart, security-aware (target) GUI model. We proved the correctness of the proposed transformation based on functions that we first constructed formally and then implemented using OCL. We implemented the whole approach using the Operational QVT transformation engine that is provided within the M2M Project, a subproject of the Eclipse Modeling Framework project.
We have presented a simple but extensible modeling language for GUI models that owns all the properties needed for transforming them into smart, security-aware GUI models preserving the access-control policies defined on the underlying data-model and we defined also a SecureUML dialect for GUI models. In this context, we defined security-awareness and introduced a methodology for automatically analyzing the security properties of GUI models w.r.t. the corresponding protected data model. Further on, we refined the approach taken to introduce our definition of smartness. We also provided functions to compute the access control information for each GUI element according to the underlying security policy protecting the data model. We implemented these functions in QVT so we automate the computing process to obtain smart, security aware GUIs. We proved that the model-to-model transformation is such that: it only generate smart GUIs and preserves security-awareness: that is, what was forbidden (actions on the ComponentUML resources) before, is still forbidden; and what was allowed (actions on the ComponentUML resources) is still allowed. Finally, we provided a method to validate the definition of smart, security-aware GUIs by checking the corresponding OCL expressions over the SecureUML+GUI models

that capture the smart, security aware GUI definitions

Our model-transformation based approach for designing smart, 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 or know at all. 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 smart, security-aware GUI models to check that they are designing the right GUI to give the authorized users access to the intended application data. A crucial property of these systems that we address with our model-transformation based approach, is *compliance*. For the case considered in this thesis, compliance means that executing events on the GUI layer never leads to program exceptions from the access control security policy implemented at the persistence layer.
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 system layers as part of a security-intensive industrial software development.

6.2 Future Work

The work presented here rises questions that deserve future research efforts. The variety of extensions of the proposed methodologies to automatically generating smart, security-aware GUIs are manifold.
For example, our work is related to research in the intelligent user interface field. In our view, smart, security-aware GUIs can be seen as a class of intelligent user interfaces: they 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 to generating other classes of intelligent interfaces. In particular, it would be also interesting to use the proposed methodologies for generating other concepts of smart, security-aware GUIs, possibly giving the developer, at the end, the chance to tailor his own interpretation of smartness. That is, to influence the lifting of the permissions.
Another interesting extension of this work is the development of analyzing tools for the GUI designer. For example a GUI designer could model the GUI, transform it into a smart, security-aware GUI and then use an analyzing tool for visually checking the reachability of the GUI elements for specific roles or even specific users, taking into account the authorization constraints. From a different perspective, we could also build a tool where we require an interface designed for a role that users in this role can reach a predefined bunch of GUI elements, what should be validated in the model and in the actual GUI implementation later on.

Another open point is the extension of the GUI modeling language proposed in this work. We already described briefly how extensions can be done and gave some examples. For instance, in the case study presented we already assumed a little extension. Probably, when integrating the security transformations into an integrated development environment together with a rapid application development tool, the language will have to be modified accordingly to meet platform specific requirements. Also, considering the integration of other system layers in the approach would make it interesting for other domain experts and perhaps more reusable. All of these possible extensions are aimed to help the GUI designer to build better, faster and more reliable GUIs.

Last but not least, code generation within the model driven security initiative deserves a closer attention. There is a huge range of possibilities and approaches to design multiple system layers from models to code. Naturally, the next upcoming step of our approach is to actually generate a running implementation of the derived models. We

could try to manually tailor some tools to support our approach but code generation fits better in our setting within MDA and have also a long list of theoretical advantages. Thus, we present next our considerations within MDA about the possibility of automatically generate code reflecting the outcomes of this thesis.

Code Generation for Smart, Security-aware GUIs

Code generation is the technique of writing and using programs that build application and system code. These programs take a declarative representation of the design of the code and deliver as an output one or more target files, e. g. Java code, SQL or other controlled output. Figure 6.1 shows a general overview of the process of code generation. A *code generator* takes as an input the design representation and one or several *templates* and automatically builds the output code. The design describes the application to build and the templates define the interpretation of the design elements for the specific platform. A platform in this context means a concrete environment in which the code is being executed, e. g. .NET or J2EE.

The question of whether code generation from models is an actual possibility is very discussed within the MDA community but still sustained as a principal goal to be accomplished for some argued reasons: the quality of the code improves since a program "writes" the code, thus decreasing errors and inconsistencies possibly introduced by a developer doing everything by hand; final productivity is increased; the code can be re-generated with ease; the design can be specified in an abstract form uncoupled from implementation details; .... Besides these benefits there are as well drawbacks. First of all, today's generators work not particularly good when integrated within an existing code base. As the design models doesn't take into account platform specific properties and behaviors, the capabilities of designing platform-specific applications is limited.

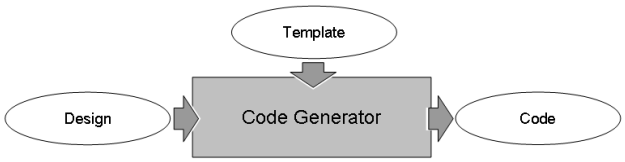

Figure 6.1: The process of code generation.

Although the set of standards provided by MDA are still rather conceptual at the moment, meaning that they leave open many points for interpretation, they are a good starting point to agree on and further improve the concepts. As many solutions (OptimalJ, AndroMDA, MDE, ...) exist implementing this approach, it might be interesting to consider them for generating GUI code together with the needed access-control infrastructures, from our platform independent smart, security-aware GUI models.
A crucial point in this context will be the actual implementation of an access-control infrastructure. As this depends on the specific target platform one has to consider several problems in this domain. Following we state some of them.

Firstly the organization of the access-control infrastructure has to be discussed. In [3] the authors provide examples for the generation of such infrastructures for distributed systems. There the access-control information is generated into deployment-descriptors and directly in the code as guards of functions. The drawback of this methodology is, that whenever the security-requirements change, the code has to be re-generated as

well. We propose here to further develop a centralized solution in which the access-control information is not directly hard-coded into the source code but in a kind of an access control database. The GUI widgets and the corresponding event listeners can be indexed by an identifier, when a user requests access on such a widget or its corresponding events, a centralized security-monitor checks whether the user is permitted or not according to the context information on the one hand and the permission assignment stored in the proposed database on the other hand. Whenever the security-requirements change, not the GUI code itself has to be re-generated but only the access-control database. One might argue, that the idea of a centralized security-monitor would decrease the system performance, this could be circumvented by considering a hybrid approach caching the access-control information in the context of a widget and periodically updating this information from the access-control database.

Another open point is how to derive on a clients GUI the run-time context information of the possibly distributed server application in order to check the authorization constraints. A possible solution could be to provide a centralized server-side security-service in such settings to which the client applications have access.

However, besides these interesting open points there are many more waiting for being discovered in this context in order to have at the end concrete tool support for making model-driven security an easy to use integrated methodology for developing security-intensive applications on multiple system layers.

Appendix A

Aggregation Transformation in Detail

In this appendix we provide the full description of the Aggregation Transformation introduced and described in chapters 4 and 5, using the Operational QVT syntax [15].

A.1 The main function of the transformation

```
modeltype GUI uses "http://gui/1.0";
modeltype SECUMLANDCOMPUML uses "http://secureumlandcomponentuml/1.0";
modeltype SECUMLANDGUI uses "http://secureumlandgui/1.0";

transformation AggregationTransformation(in guiModel : GUI,
  in secModel : SECUMLANDCOMPUML, out guiSecModel : SECUMLANDGUI);

main() {
  /* Copy of the whole SECUMANDCOMPUML model into the output model */
  -- Copy the Users
  secModel.objects()[SECUMLANDCOMPUML::User]->map User_to_User();
  -- Copy the Roles (with hierarchy!)
  secModel.objects()[SECUMLANDCOMPUML::Role]->map Role_to_Role();
  secModel.objects()[SECUMLANDCOMPUML::Role]->preserveRoleHierarchy();
  -- Copy the Permissions
  secModel.objects()[SECUMLANDCOMPUML::Permission]->map Permission_to_Permission();
  -- Copy the AuthorizationConstraints
  secModel.objects()[SECUMLANDCOMPUML::AuthorizationConstraint]->map
    AuthConst_to_AuthConst();
  -- Copy the Actions (with hierarchy!)
  secModel.objects()[SECUMLANDCOMPUML::AtomicRead]->map Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::AtomicUpdate]->map Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::AtomicCreate]->map Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::AtomicDelete]->map Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::AtomicExecute]->map Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::EntityRead]->map Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::EntityUpdate]->map Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::EntityFullAccess]->map Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::AttributeFullAccess]->map Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::AssociationEndFullAccess]->map
    Action_to_Action();
  secModel.objects()[SECUMLANDCOMPUML::CompositeAction]->preserveActionHierarchy();
  -- Copy the Entities
  secModel.objects()[SECUMLANDCOMPUML::Entity]->map Resource_to_Resource();
```

```
  -- Copy the Attributes
  secModel.objects()[SECUMLANDCOMPUML::Attribute]->map Resource_to_Resource();
  -- Copy the Methods
  secModel.objects()[SECUMLANDCOMPUML::Method]->map Resource_to_Resource();
  -- Copy the AssociationEnds
  secModel.objects()[SECUMLANDCOMPUML::AssociationEnd]->map Resource_to_Resource();
  -- Copy the Association
  secModel.objects()[SECUMLANDCOMPUML::Association]->map Association_to_Association();

  /* Copy of the whole GUI model into the output model */
  -- First the all the Widgets
  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();
  -- Then all the Events
  guiModel.objects()[GUI::CompEvent]->map CompEvent_to_CompEvent();
  guiModel.objects()[GUI::Event]->map Event_to_Event();
  -- Then all the Actions
  guiModel.objects()[GUI::WidgetAction]->map GAction_to_GAction();
  -- The "glueing" step of this aggregation, i.e. combining the GUI model part
  -- with the SecureUML+ComponentUML part
  guiModel.objects()[GUI::ModelAction]->map GAction_to_GAction();
  -- Now the introduction of the onCreateEvents
  guiSecModel.objects()[SECUMLANDGUI::Widget]->map addOnCreateEvent();
  -- Initialization of the Container property (onCreate opens all contained Widgets)
  guiSecModel.objects()[SECUMLANDGUI::Container]->map initializeContainerProperty();
  -- Last but not least, we create an AtomicExecute Action for every Event
  guiSecModel.objects()[SECUMLANDGUI::Event]->map addAtomicExecuteAction();
}
```

A.2 The mapping functions for the security model

```
-- Mapping of the Users
mapping SECUMLANDCOMPUML::User::User_to_User() : SECUMLANDGUI::User
{
  name := self.name;
}

-- Mapping of the Roles
mapping SECUMLANDCOMPUML::Role::Role_to_Role() : SECUMLANDGUI::Role
{
  name := self.name;
  default := self.default;
  includes := self.includes.resolve()->oclAsType(SECUMLANDGUI::User)->asOrderedSet();
}

-- Preservation of the Role hierarchy
mapping inout SECUMLANDCOMPUML::Role::preserveRoleHierarchy()
{
  init{
    var mappedRole := self.resolveone().oclAsType(SECUMLANDGUI::Role);
    var mappedSubRoles := self.subRoles.resolve().oclAsType(SECUMLANDGUI::Role)->
      asOrderedSet();
    mappedRole.subRole := mappedSubRoles;
  }
}

-- Mapping of the Permissions
mapping SECUMLANDCOMPUML::Permission::Permission_to_Permission() :
```

```
  SECUMLANDGUI::Permission
{
  name := self.name;
  default := self.default;
  givesAccess := self.givesAccess.resolveone().oclAsType(SECUMLANDGUI::Role);
}

-- Mapping of the AuthorizationConstraints
mapping SECUMLANDCOMPUML::AuthorizationConstraint::AuthConst_to_AuthConst() :
  SECUMLANDGUI::AuthorizationConstraint
{
  -- name := self.name;
  language := self.language;
  _body := self._body;
  constrains := self.constrains.resolveone().oclAsType(SECUMLANDGUI::Permission);
}

-- Mapping of the Actions
mapping SECUMLANDCOMPUML::AtomicRead::Action_to_Action() :
  SECUMLANDGUI::AtomicRead
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::AtomicUpdate::Action_to_Action() :
  SECUMLANDGUI::AtomicUpdate
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::AtomicCreate::Action_to_Action() :
  SECUMLANDGUI::AtomicCreate
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::AtomicDelete::Action_to_Action() :
  SECUMLANDGUI::AtomicDelete
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::AtomicExecute::Action_to_Action() :
  SECUMLANDGUI::AtomicExecute
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::EntityRead::Action_to_Action() :
  SECUMLANDGUI::EntityRead
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::EntityUpdate::Action_to_Action() :
  SECUMLANDGUI::EntityUpdate
```

```
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::EntityFullAccess::Action_to_Action() :
  SECUMLANDGUI::EntityFullAccess
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::AttributeFullAccess::Action_to_Action() :
  SECUMLANDGUI::AttributeFullAccess
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::AssociationEndFullAccess::Action_to_Action() :
  SECUMLANDGUI::AssociationEndFullAccess
{
  name := self.name;
  isAssigned := self.isAssigned.resolve()->oclAsType(SECUMLANDGUI::Permission)->
    asOrderedSet();
}

-- Preservation of the Action hierarchy
mapping inout SECUMLANDCOMPUML::CompositeAction::preserveActionHierarchy()
{
  init{
    var mappedAction := self.resolveone().oclAsType(SECUMLANDGUI::CompositeAction);
    var mappedSubActions := self.subordinatedActions.resolve().
      oclAsType(SECUMLANDGUI::Action)->asOrderedSet();
    mappedAction.subordinatedActions := mappedSubActions;
  }
}

-- Mapping of the Resources
mapping SECUMLANDCOMPUML::Entity::Resource_to_Resource() :
  SECUMLANDGUI::Entity
{
  name := self.name;
  actions := self.actions.resolve()->oclAsType(SECUMLANDGUI::Action)->
    asOrderedSet();
}
mapping SECUMLANDCOMPUML::Attribute::Resource_to_Resource() :
  SECUMLANDGUI::Attribute
{
  name := self.name;
  actions := self.actions.resolve()->oclAsType(SECUMLANDGUI::Action)->
    asOrderedSet();
  entityAttribute := self.entityAttribute.resolveone().
    oclAsType(SECUMLANDGUI::Entity);
}
mapping SECUMLANDCOMPUML::Method::Resource_to_Resource() :
  SECUMLANDGUI::Method
{
  name := self.name;
  isQuery := self.isQuery;
  actions := self.actions.resolve()->oclAsType(SECUMLANDGUI::Action)->
    asOrderedSet();
```

```
  entityMethod := self.entityMethod.resolveone().
    oclAsType(SECUMLANDGUI::Entity);
}
mapping SECUMLANDCOMPUML::AssociationEnd::Resource_to_Resource() :
  SECUMLANDGUI::AssociationEnd
{
  name := self.name;
  actions := self.actions.resolve()->oclAsType(SECUMLANDGUI::Action)->
    asOrderedSet();
  entityAssociationEnd := self.entityAssociationEnd.resolveone().
    oclAsType(SECUMLANDGUI::Entity);
}

mapping SECUMLANDCOMPUML::Association::Association_to_Association() :
  SECUMLANDGUI::Association
{
  associationEnd := self.associationEnd.resolve()->
    oclAsType(SECUMLANDGUI::AssociationEnd)->asOrderedSet();
}
```

A.3 The mapping functions for the gui model

```
-- Mapping of the Windows
mapping GUI::Window::Widget_to_Widget() : SECUMLANDGUI::Window
{
  label := self.label;
}
-- Mapping of the Entries
mapping GUI::Entry::Widget_to_Widget() : SECUMLANDGUI::Entry
{
  label := self.label;
}
-- Mapping of the Buttons
mapping GUI::Button::Widget_to_Widget() : SECUMLANDGUI::Button
{
  label := self.label;
}

-- Preserving the containment hierarchy
mapping inout GUI::Widget::preserve_containment_hierarchy()
when{self.oclIsKindOf(GUI::Container)}
{
  init{
    var mappedContainer := self.resolveone().oclAsType(SECUMLANDGUI::Container);
    var mappedContainedWidgets := self.oclAsType(GUI::Container).contained.
      resolve().oclAsType(SECUMLANDGUI::Widget)->asOrderedSet();
    mappedContainer.contained := mappedContainedWidgets;
  }
}

-- Mapping of the events of the gui model
mapping GUI::CompEvent::CompEvent_to_CompEvent() : SECUMLANDGUI::CompEvent
{
  name := self.name; -- Don't need, just for debugging!
  event := self.event.getEventEnumLiteral();
  holder := self.holder.resolveone().oclAsType(SECUMLANDGUI::Widget);
}
mapping GUI::Event::Event_to_Event() : SECUMLANDGUI::Event
when{self.oclIsTypeOf(GUI::CompEvent).not()}
{
  name := self.name; -- Don't need, just for debugging!
```

```
  event := self.event.getEventEnumLiteral();
  holder := self.holder.resolveone().oclAsType(SECUMLANDGUI::Widget);
}
-- Returns the corresponding Instance of the Event enumeration of the output model
query GUI::EventEnum::getEventEnumLiteral() : SECUMLANDGUI::EventEnum
{
  if (self = GUI::EventEnum::onEnter) then {
    return SECUMLANDGUI::EventEnum::onEnter;
  }
  endif;
  if (self = GUI::EventEnum::onLeave) then {
    return SECUMLANDGUI::EventEnum::onLeave;
  }
  endif;
  if (self = GUI::EventEnum::onClick) then {
    return SECUMLANDGUI::EventEnum::onClick;
  }
  endif;
  if (self = GUI::EventEnum::onDoubleClick) then {
    return SECUMLANDGUI::EventEnum::onDoubleClick;
  }
  endif;
}

-- Mapping of the WidgetActions of the gui model
mapping GUI::WidgetAction::GAction_to_GAction() : SECUMLANDGUI::WidgetAction
{
  name := self.name; -- Don't need, just for debugging!
  guiAction := self.guiAction.getGUIActionLiteral();
  triggeredBy := self.triggeredBy.resolveone().oclAsType(SECUMLANDGUI::Event);
  actionOn := self.actionOn.resolveone().oclAsType(SECUMLANDGUI::Widget);
}

-- Returns the corresponding Instance of the Action enumeration of the output model
query GUI::GUIAction::getGUIActionLiteral() : SECUMLANDGUI::GUIAction
{
  if (self = GUI::GUIAction::open) then {
    return SECUMLANDGUI::GUIAction::open;
  }
  endif;
  if (self = GUI::GUIAction::close) then {
    return SECUMLANDGUI::GUIAction::close;
  }
  endif;
}

-- Mapping of the ModelActions of the gui model
mapping GUI::ModelAction::GAction_to_GAction() : SECUMLANDGUI::ModelAction
{
  name := self.name; -- Don't need, just for debugging!
  triggeredBy := self.triggeredBy.resolveone().oclAsType(SECUMLANDGUI::Event);
  modelAction := SECUMLANDGUI::AtomicAction.allInstances()->
    select(a | self.modelAction.name->includes(a.name))->asOrderedSet()->first();
}

-- Adding an onCreateEvent to every Widget
mapping SECUMLANDGUI::Widget::addOnCreateEvent() : SECUMLANDGUI::CompEvent
{
  name := self.label + 'OnCreateEvent';
  holder := self;
  event := SECUMLANDGUI::EventEnum::onCreate;
}
```

```
-- Initializing the Container property for every Container
mapping SECUMLANDGUI::Container::initializeContainerProperty() :
  Set(SECUMLANDGUI::WidgetAction)
{
  init{
    result := self.contained->map createWidgetAction(self.widgetEvents->
      select(e | e.event = SECUMLANDGUI::EventEnum::onCreate)->asOrderedSet()->
      first())->asSet();
  }
}

-- Creating a new (open) WidgetAction fired by the given Event on the given Widget
mapping SECUMLANDGUI::Widget::createWidgetAction(ev : SECUMLANDGUI::Event) :
  SECUMLANDGUI::WidgetAction
{
  name := ev.holder.label + 'onCreateOpenActionOn' + self.label;
  triggeredBy := ev;
  actionOn := self;
  guiAction := SECUMLANDGUI::GUIAction::open;
}

-- Adding an AtomicExecuteAction to the Events
mapping SECUMLANDGUI::Event::addAtomicExecuteAction() : SECUMLANDGUI::AtomicExecute
{
  -- Simple composition of Strings for the name
  name := self.holder.label + self.event.repr() + 'AtomicExecute';
  -- Referencing the corresponding resource
  resource := self;
}
```

A.4 Preservation checks

We implemented QVTO queries for checking the correctness of the transformation, the full implementation of these queries can be found in the file *AggregationTransformation.qvto* on the attached CD.

Appendix B

Security Transformation in Detail

In this appendix we provide the full description of the Security Transformation introduced and described in chapter 4 , using the Operational QVT syntax [15]. The running full implementation provided together with the example introduced in chapter 2 is provided on the CD.

B.1 The main function of the transformation

```
modeltype SECUMLANDGUI uses "http://secureumlandgui/1.0";

transformation SecurityTransformation(inout secGUI : SECUMLANDGUI);

main() {
secGUI.objectsOfType(Event)->map liftPermissions();
}
```

B.2 The actual mapping functions

```
/* Get the allowed Roles and setup a new Permission for each of them to
    execute the given Event */
mapping Event::liftPermissions() : Set(Permission)
{
  init{
    result := self.AllowedRolesG()->setPermission(self)->asSet();
  }
}

/* Creation of a new Permission for the given Role on the AtomicExecute
    Action of the given Event */
mapping Role::setPermission(e: Event) : Permission
{
  name := self.name + e.holder.label + e.event.repr() + 'Execute';
  accesses := e.actions;
  default := false;
  givesAccess := self;
  isConstraintBy := e.map setAuthConst(self);
}

/* Creation of a new AuthorizationConstraint for the above defined Permission
```

```
    using the function ConstraintG() */
mapping Event::setAuthConst(rl : Role) : AuthorizationConstraint
{
  language := 'Some language';
  _body := self.ConstraintG(rl);
}
```

B.3 $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$

```
query Event::AllowedRolesG() : Set(Role)
{
  if(self.DaAc()->isEmpty()) then{
    return Role.allInstances();
  }
  else{
    return setIntersection(
      self.DaAc().modelAction->collectNested(daac | daac.DaAu())->asSet()
    );
  }
  endif;
}

query Event::ConstraintG(rl : Role) : String
{
  if(self.DaAc()->isEmpty()) then{
    return 'TRUE';
  }
  else{
    return setConjunction(
      self.DaAc().modelAction->collect(daac | daac.AuthConstA(rl))->asSet()
    );
  }
  endif;
}
```

B.4 Model queries

```
-- Returns all the Events of a given Widget
query Widget::Ev() : Set(Event)
{
  return self.widgetEvents;
}

-- Returns all the compulsory Events of a given Widget
query Widget::compEv() : Set(CompEvent)
{
  return self.widgetEvents->select(e | e.oclIsTypeOf(CompEvent))->
    oclAsType(CompEvent)->asSet();
}

-- Returns all the Widgets contained by the given Container
query Container::In() : Set(Widget)
{
  return self.contained;
}

-- Returns all the WidgetActions fired by the given Event
query Event::WdAc() : Set(WidgetAction)
{
  return self.firedActions->select(a | a.oclIsTypeOf(WidgetAction))->
```

```
    oclAsType(WidgetAction)->asSet();
}

-- Returns all the ModelActions fired by the given Event
query Event::DaAc() : Set(ModelAction)
{
  return self.firedActions->select(a | a.oclIsTypeOf(ModelAction))->
    oclAsType(ModelAction)->asSet();
}

-- Returns all the Actions fired by the given Event
query Event::Ac() : Set(GAction)
{
  return self.DaAc().oclAsType(GAction)->asSet()->
    union(self.WdAc().oclAsType(GAction)->asSet());
}

-- Returns the set of Roles that is allowed to execute the given Event
query Event::EvAu() : Set(Role)
{
  return Role.allInstances()->select(r | r.allPermissions().accesses->
    includesAll(self.actions));
}

-- Returns the AuthorizationConstraint body of a given Role on a given Event
query Event::AuthConstG(rl : Role) : String
{
return self.actions.isAssigned->select(p | p.givesAccess = rl)->asOrderedSet()->
  first().isConstraintBy._body;
}

-- Returns the set of Roles that is allowed to perform the given data model action
query Action::DaAu() : Set(Role)
{
return Role.allInstances()->select(r | r.allPermissions().accesses.
  subactionPlus()->includesAll(self.subactionPlus()));
}

-- Returns the combined AuthorizationConstraint
query AtomicAction::AuthConstA(rl : Role) : String
{
return setDisjunction(Permission.allInstances()->select(p |
  p.accesses.subactionPlus()->includes(self) and rl.allPermissions()->
  includes(p)).isConstraintBy._body->asSet());
}

-- Returns the intersection of the sets of roles given
query setIntersection(rs: Set(Set(Role))) : Set(Role)
{
if(rs->size() = 1) then{
return rs->asOrderedSet()->first();
}
else{
return rs->asOrderedSet()->first()->intersection(
setIntersection(rs->excluding(rs->asOrderedSet()->first()))
);
}
endif;
}

-- Returns the conjunction of the authorization constraints in String representation
query setConjunction(acs : Set(String)) : String
```

```
{
if(acs->size() = 1) then{
return acs->asOrderedSet()->first();
}
else{
return '(' + acs->asOrderedSet()->first() + ') and (' + setConjunction(
acs->excluding(acs->asOrderedSet()->first())
) + ')';
}
endif;
}

-- Returns the disjunction of the authorization constraints in String representation
query setDisjunction(acs : Set(String)) : String
{
if(acs->size() = 1) then{
return acs->asOrderedSet()->first();
}
else{
return '(' + acs->asOrderedSet()->first() + ') or (' + setDisjunction(
acs->excluding(acs->asOrderedSet()->first())
) + ')';
}
endif;
}

-- Returns the set of AtomicActions included by a given Action in Action hierarchy
query Action::subactionPlus() : Set(AtomicAction)
{
if (self.oclIsKindOf(AtomicAction)) then {
return Set {self.oclAsType(AtomicAction)};
}
else {
return self.oclAsType(CompositeAction).subordinatedActions.
  subactionPlus()->asSet();
}
endif;
}

-- Returns all the assigned Permissions of a Role including the Role hierarchy
query Role::allPermissions() : Set(Permission)
{
return self.superrolePlus().hasPermission->asSet();
}

-- Returns all the superroles of a Role in the Role hierarchy
query Role::superrolePlus() : Set(Role)
{
return self.superrolePlusOnSet(self.superRole);
}

-- Returns the set of superroles on a set of Roles
query Role::superrolePlusOnSet(rs : Set(Role)) : Set(Role)
{
if (rs.superRole->exists(r | rs->excludes(r))) then {
return self.superrolePlusOnSet(rs->union(rs.superRole)->asSet());
}
else {
return rs->including(self);
}
endif;
}
```

B.5 Security-Awareness property check

The following queries can be used to check whether a given GUI model is security-aware or not.

```
query Event::EvAuSecAware() : Boolean
{
  return not self.DaAc()->isEmpty() implies self.EvAu() = setIntersection(
    self.DaAc().modelAction->collectNested(daac | daac.DaAu())->asSet()
  );
}

query Event::AuthConstGSecAware() : Boolean
{
  return not self.DaAc()->isEmpty() implies self.EvAu()->forAll(rl |
    self.AuthConstG(rl)->asOrderedSet() = setConjunction(
      self.DaAc().modelAction->AuthConstA(rl)->asSet())->asOrderedSet()
  );
}
```

Appendix C

SmartSec Transformation in Detail

In this appendix we provide the full description of the SmartSec Transformation introduced and described in chapter 5 , using the Operational QVT syntax [15]. We will not repeat here the queries and mappings that are similar as in Appendix B. The running full implementation provided together with the example introduced in chapter 2 is provided on the CD. As the main function and the mapping functions remain the same, we will concentrate on the implementation of the functions $AllowedRoles_{\mathcal{G}}$ and $Constraint_{\mathcal{G}}$. These functions compute the in the case of smartness other access-control information then in the case of pure security-awareness. The implementation follows straight the constructions proposed in section 5.5.

C.1 Function $AllowedRoles_{\mathcal{G}}$

```
query Event::AllowedRolesG() : Set(Role)
{
  if((self.event = EventEnum::onCreate).not()) then{
    if(self.DaAc()->isEmpty() and self.WdAc()->select(a | a.guiAction =
      GUIAction::open)->isEmpty()) then{
      return Role.allInstances();
    }
    else{
      if((self.DaAc()->isEmpty()).not() and self.WdAc()->select(a | a.guiAction =
        GUIAction::open)->isEmpty()) then{
        return setIntersection(
          self.DaAc().modelAction->collectNested(daac | daac.DaAu())->asSet()
        );
      }
      else{
        if(self.DaAc()->isEmpty() and (self.WdAc()->select(a | a.guiAction =
          GUIAction::open)->isEmpty()).not()) then{
          return setIntersection(
            self.WdAc().actionOn->collectNested(wd | wd.Ev()->select(e | e.event =
              EventEnum::onCreate)->asOrderedSet()->first().AllowedRolesG())->
              asSet()
          );
        }
        else{
          return setIntersection(
            self.DaAc().modelAction->collectNested(daac | daac.DaAu())->asSet()
```

```
          )->intersection(
            setIntersection(
              self.WdAc().actionOn->collectNested(wd | wd.Ev()->select(e |
                e.event = EventEnum::onCreate)->asOrderedSet()->
                first().AllowedRolesG())->asSet()
            )
          );
        }
        endif;
      }
      endif;
    }
    endif;
  }
  else{
    if((self.holder.oclIsKindOf(Container)).not() and self.holder.Ev()->
      excluding(self.holder.Ev()->select(e | e.event = EventEnum::onCreate)->
        asOrderedSet()->first())->isEmpty()) then{
      return Role.allInstances();
    }
    else{
      if((self.holder.oclIsKindOf(Container)).not() and
        (self.holder.Ev()->excluding(self.holder.Ev()->select(e | e.event =
          EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()).not() and
        self.holder.compEv()->excluding(self.holder.compEv()->select(e | e.event =
          EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()) then{
        return self.holder.Ev()->excluding(self.holder.Ev()->select(e | e.event =
          EventEnum::onCreate)->asOrderedSet()->first())->
            collect(e | e.AllowedRolesG())->asSet();
      }
      else{
        if((self.holder.oclIsKindOf(Container)).not() and
          (self.holder.compEv()->excluding(self.holder.compEv()->select(e |
            e.event = EventEnum::onCreate)->asOrderedSet()->first())->
            isEmpty()).not()
          ) then{
          return setIntersection(
            self.holder.compEv()->excluding(self.holder.compEv()->select(e |
              e.event = EventEnum::onCreate)->asOrderedSet()->first())->
                collectNested(ev | ev.AllowedRolesG())->asSet()
          );
        }
        else{
          if(self.holder.oclIsKindOf(Container)) then{
  return setIntersection(
              self.holder.oclAsType(Container).In()->collect(wd | wd.Ev()->
                select(e | e.event = EventEnum::onCreate))->collectNested(ev |
                  ev.AllowedRolesG())->asSet()
            );
          }
endif;
        }
        endif;
      }
      endif;
    }
    endif;
  }
  endif;
}
```

C.2 Function *Constraint*$_G$

```
query Event::ConstraintG(rl : Role) : String
{
  if((self.event = EventEnum::onCreate).not()) then{
    if(self.DaAc()->isEmpty() and self.WdAc()->select(a | a.guiAction =
      GUIAction::open)->isEmpty()) then{
      return 'TRUE';
    }
    else{
      if((self.DaAc()->isEmpty()).not() and self.WdAc()->select(a | a.guiAction =
        GUIAction::open)->isEmpty()) then{
        return setConjunction(
          self.DaAc().modelAction->collect(daac | daac.AuthConstA(rl))->asSet()
        );
      }
      else{
        if(self.DaAc()->isEmpty() and (self.WdAc()->select(a | a.guiAction =
          GUIAction::open)->isEmpty()).not()) then{
          return setConjunction(
            self.WdAc().actionOn->collect(wd | wd.Ev()->select(e | e.event =
              EventEnum::onCreate)->asOrderedSet()->first().ConstraintG(rl))->
              asSet()
          );
        }
        else{
          return '(' + setConjunction(
            self.DaAc().modelAction->collect(daac | daac.AuthConstA(rl))->
            asSet()) + ') and (' +
          setConjunction(
            self.WdAc().actionOn->collect(wd | wd.Ev()->select(e | e.event =
              EventEnum::onCreate)->asOrderedSet()->first().ConstraintG(rl))->
              asSet()) + ')';
        }
        endif;
      }
      endif;
    }
    endif;
  }
  else{
    if((self.holder.oclIsKindOf(Container)).not() and self.holder.Ev()->
      excluding(self.holder.Ev()->select(e | e.event = EventEnum::onCreate)->
      asOrderedSet()->first())->isEmpty()) then{
      return 'TRUE';
    }
    else{
      if((self.holder.oclIsKindOf(Container)).not() and
        (self.holder.Ev()->excluding(self.holder.Ev()->select(e | e.event =
        EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()).not() and
        self.holder.compEv()->excluding(self.holder.compEv()->select(e | e.event =
        EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()) then{
        return setDisjunction(
          self.holder.Ev()->excluding(self.holder.Ev()->select(e | e.event =
            EventEnum::onCreate)->asOrderedSet()->first())->collect(e |
              e.ConstraintG(rl))->asSet()
        );
      }
      else{
        if((self.holder.oclIsKindOf(Container)).not() and
          (self.holder.compEv()->excluding(self.holder.compEv()->select(e |
            e.event = EventEnum::onCreate)->asOrderedSet()->first())->
```

```
          isEmpty()).not()
        ) then{
        return setConjunction(
   self.holder.compEv()->excluding(self.holder.compEv()->select(e |
     e.event = EventEnum::onCreate)->asOrderedSet()->first())->
     collect(ev | ev.ConstraintG(rl))->asSet()
        );
      }
      else{
        if(self.holder.oclIsKindOf(Container)) then{
          return setConjunction(
            self.holder.oclAsType(Container).In()->collect(wd | wd.Ev()->
              select(e | e.event = EventEnum::onCreate))->collect(ev |
              ev.ConstraintG(rl))->asSet()
          );
        }
        endif;
      }
      endif;
    }
    endif;
  }
  endif;
}
```

C.3 Smartness properties check

The following queries can be used to check whether a given GUI model is security-aware or not.

```
query Event::EvAuSmartSecAware_1() : Boolean
{
  return self.EvAu() = self.SecAware()->intersection(self.Smart());
}

query Event::SecAware() : Set(Role)
{
  if(self.DaAc()->isEmpty()) then{
    return Role.allInstances();
  }
  else{
    return setIntersection(
      self.DaAc()->modelAction->collectNested(daac | daac.DaAu())->asSet()
    );
  }
  endif;
}

query Event::Smart() : Set(Role)
{
  if(self.WdAc()->select(a | a.guiAction = GUIAction::open)->isEmpty()) then{
    return Role.allInstances();
  }
  else{
    return setIntersection(
      self.WdAc()->select(a | a.guiAction = GUIAction::open).actionOn->
        collectNested(wd | setIntersection(wd.compEv()->collectNested(ev |
        ev.EvAu())->asSet()
      ))->asSet()
    );
```

```
  }
  endif;
}

query Event::EvAuSmartSecAware_2() : Boolean
{
  if((self.holder.oclIsKindOf(Container)).not() and self.holder.Ev()->
    excluding(self.holder.Ev()->select(e | e.event = EventEnum::onCreate)->
    asOrderedSet()->first())->isEmpty()) then{
    return self.EvAu() = Role.allInstances();
  }
  else{
    if((self.holder.oclIsKindOf(Container)).not() and
      (self.holder.Ev()->excluding(self.holder.Ev()->select(e | e.event =
      EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()).not() and
      self.holder.compEv()->excluding(self.holder.compEv()->select(e | e.event =
      EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()) then{
      return self.EvAu() = self.holder.Ev()->excluding(self.holder.Ev()->select(e |
      e.event = EventEnum::onCreate)->asOrderedSet()->first())->collect(e |
      e.AllowedRolesG())->asSet();
    }
    else{
      if((self.holder.oclIsKindOf(Container)).not() and
        (self.holder.compEv()->excluding(self.holder.compEv()->select(e | e.event =
        EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()).not()
        ) then{
        return self.EvAu() = setIntersection(
          self.holder.compEv()->excluding(self.holder.compEv()->select(e | e.event =
          EventEnum::onCreate)->asOrderedSet()->first())->collectNested(ev |
          ev.AllowedRolesG())->asSet()
        );
      }
      else{
        if(self.holder.oclIsKindOf(Container)) then{
          return self.EvAu() = setIntersection(
            self.holder.oclAsType(Container).In()->collect(wd | wd.Ev()->select(e |
            e.event = EventEnum::onCreate))->collectNested(ev | ev.AllowedRolesG())->
            asSet()
          );
        }
        endif;
      }
      endif;
    }
    endif;
  }
  endif;
}


query Event::AuthConstGSmartSecAware_3() : Boolean
{
  if(self.DaAc()->isEmpty() and self.WdAc()->select(a | a.guiAction =
    GUIAction::open)->isEmpty()) then{
    return self.EvAu()->forAll(rl | self.AuthConstG(rl) = 'TRUE');
  }
  else{
    if((self.DaAc()->isEmpty()).not() and self.WdAc()->select(a | a.guiAction =
      GUIAction::open)->isEmpty()) then{
      return self.EvAu()->forAll(rl | self.AuthConstG(rl) = setConjunction(
        self.DaAc().modelAction->collect(daac | daac.AuthConstA(rl))->asSet()
      ));
```

```
    }
    else{
      if(self.DaAc()->isEmpty() and (self.WdAc()->select(a | a.guiAction =
        GUIAction::open)->isEmpty()).not()) then{
        return self.EvAu()->forAll(rl | self.AuthConstG(rl) = setConjunction(
          self.WdAc()->select(a | a.guiAction = GUIAction::open).actionOn->
          collect(wd | setConjunction(
            wd.compEv()->collect(ev | ev.AuthConstG(rl))->asSet()
          ))->asSet()
        ));
      }
      else{
        return self.EvAu()->forAll(rl | self.AuthConstG(rl) = '(' + setConjunction(
          self.DaAc().modelAction->collect(daac | daac.AuthConstA(rl))->asSet()
        ) + ') and (' + setConjunction(
          self.WdAc()->select(a | a.guiAction = GUIAction::open).actionOn->
          collect(wd | setConjunction(
            wd.compEv()->collect(ev | ev.AuthConstG(rl))->asSet()
          ))->asSet()
        ));
      }
      endif;
    }
    endif;
  }
  endif;
}

query Event::AuthConstGSmartSecAware_4() : Boolean
{
  if((self.holder.oclIsKindOf(Container)).not() and self.holder.Ev()->
    excluding(self.holder.Ev()->select(e | e.event = EventEnum::onCreate)->
    asOrderedSet()->first())->isEmpty()) then{
    return self.EvAu()->forAll(rl | self.AuthConstG(rl) = 'TRUE');
  }
  else{
    if((self.holder.oclIsKindOf(Container)).not() and
      (self.holder.Ev()->excluding(self.holder.Ev()->select(e | e.event =
      EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()).not() and
      self.holder.compEv()->excluding(self.holder.compEv()->select(e | e.event =
      EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()) then{
      return self.EvAu()->forAll(rl | self.AuthConstG(rl) = setDisjunction(
        self.holder.Ev()->excluding(self.holder.Ev()->select(e | e.event =
        EventEnum::onCreate)->asOrderedSet()->first())->collect(e |
        e.ConstraintG(rl))->asSet()
      ));
    }
    else{
      if((self.holder.oclIsKindOf(Container)).not() and
        (self.holder.compEv()->excluding(self.holder.compEv()->select(e |
        e.event = EventEnum::onCreate)->asOrderedSet()->first())->isEmpty()).not()
        ) then{
        return self.EvAu()->forAll(rl | self.AuthConstG(rl) = setConjunction(
          self.holder.compEv()->excluding(self.holder.compEv()->select(e |
          e.event = EventEnum::onCreate)->asOrderedSet()->first())->collect(ev |
          ev.ConstraintG(rl))->asSet()
        ));
      }
      else{
        if(self.holder.oclIsKindOf(Container)) then{
          return self.EvAu()->forAll(rl | self.AuthConstG(rl) = setConjunction(
            self.holder.oclAsType(Container).In()->collect(wd | wd.Ev()->
```

```
            select(e | e.event = EventEnum::onCreate))->collect(ev |
            ev.ConstraintG(rl))->asSet()
          ));
        }
        endif;
      }
      endif;
    }
    endif;
  }
  endif;
}
```

Appendix D

Informal how-to for QVTO on the eclipse platform

D.1 Overview

In order for the reader to get ready to use the implementations provided, we give in this appendix a short overview on using the eclipse platform for performing model transformations. We first provide a general overview on modeling on the eclipse platform and the some helpful tools. We provide a step-by-step introduction for a simple example. In section D.3 we show, how the provided implementations on the attached CD can be used and provide as well a step-by-step guide for importing these files into eclipse and get them run.

D.2 Simple example

The eclipse platform allows the automatic generation of own plug-ins and editors for developing applications. These comfortable tools helps to perform the implementation of own metamodels and operational QVT transformations. The general methodology using QVTO and eclipse is the following:

1. Creating an empty EMF Project
2. Creating an ecore Diagram in the empty EMF Project
3. Generating an EMF Model in the empty EMF Project
4. Generating the plug-in and the editor for the EMF Model
5. Running an Eclipse Application instance (preloaded with the generated plug-ins and editors)
6. Creating the input model(s)
7. Creating a transformation file
8. Configuring the run configurations
9. Run the transformation

D.2.1 Installation

In order to be able to perform the steps mentioned above, eclipse with all the necessary plug-ins have to be downloaded from the eclipse website. The easiest way is to download the whole package called *Eclipse Modeling Tools* from [25]. In this package everything needed is already preinstalled.

D.2.2 Metamodel Creation

As a first step create an empty EMF Project. Then create into this project a new Ecore Diagram. This will open a graphical editor for defining the metamodel graphically. The metamodels should have a root object with containment relation to the model-objects we would like to insert then later using the model creation wizard (See subsection Creation of Input models below). Then save the metamodel.

As a next step, create a new EMF Model in the project. This file has to be named by the suffix .genmodel. After clicking next, choose Ecore Model, then click next. Browse the workspace and choose the created .ecore file, that we have created in the last step. Now press again next and in the next window the button Finish. A new EMF Model based on the graphical metamodel is being generated.

D.2.3 Plug-in and Editor Creation

If it is not already open, open the generated EMF Model. Right-Click on the root entry and choose *Generate All*. This generates all the things needed in order to go on. A plug-in and an editor is being created, that can be used further on.

One can create several metamodels and plug-ins like described above. I recommend to create for every new metamodel a new empty EMF Project!

Now, as the plug-ins and editors are all created, we can choose *Run as...→Eclipse Application*. This runs a new second instance of eclipse, automatically loading all the defined plug-ins and editors.

D.2.4 Creation of Input models

As a first step we need to define a new Project in the second eclipse instance. Choose *Model to Model Transformation → Operational QVT Project* to create a new QVTO Project. This creates a new Project with a folder transforms in it by default. There we can put .qvto files in to execute.

As a next step we have to define the input model(s). I suggest to create a new folder called models. Then choose *new→other ...→Example EMF Model Creation Wizard→modelname Model*. Specify a name and the location to be stored. Then specify the root object of the model and store it. A new file of type .modelname has been created. With right-click on the root object we can include more elements into the model. By using the properties view at the bottom, the inserted objects can be initialized.

D.2.5 Creation of Transformations

Now, that we have the metamodels and the input model as an instance. We can create a transformation file. Create a new Operational QVT Transformation in the folder transforms. Now specify the usage of modeltypes and the transformation:

```
modeltype BOOK uses 'http://book/1.0';
modeltype PUB uses 'http://book/1.0';

transformation NewTransformation(in book : BOOK, out publication : PUB);

main() {
  book.objectsOfType(Book)->map book_to_pub();
}

mapping BOOK::Book::book_to_pub() : PUB::Publication
{
  title := self.title;
  nbPages := self.chapters->nbPages->sum();
}
```

D.2.6 Run Configurations

Now we are ready for the first transformation. Therefore, we have to specify, which models that have to transformed to which output models. Right-click on the transformation file. Choose *Run As→Run Configurations...* in the window choose *Operational QVT Interpreter* and create a new Configuration. In this new configuration we specify the input model (models, when there has been more then one input model defined in the transformation()) and the output model. Click apply and then run the transformation.

D.3 Case study

Following we refer to the case study introduced in chapter 2 in the thesis. We implemented the full case study using the concepts described above. To get started with the case study and especially with the implementations of the security transformations, we provide now a step-by-step guide.

D.3.1 Provided files

Beside the current version of eclipse on which the implementations have been tested, we provide you with the following files on the CD corresponding to the explanations done in this work.[1]

GUI metamodel: *GUI.ecore* file corresponding to figure D.1.

SecureUML+ComponentUML metamodel: *SecureUMLandComponentUML.ecore* file corresponding to figure D.2.

SecureUML+GUI Metamodel: *SecureUMLandGUI.ecore* file corresponding to figure D.3.

A sample GUI Model: *GuiModel.gui* file corresponding to the case study presented in chapter 2.

A sample SecureUML+ComponentUML Model: *SecurityPolicy.secureumlandcomponentuml* file corresponding as well to chapter 2.

Aggregation transformation file: The *AggregationTransformation.qvto* described in chapters 4 and 5.

[1] Note, that the figures of the metamodels are contained on the CD as well, for readability reasons.

Security transformation file: The *SecurityTransformation.qvto* described in chapter 4.

Smartness transformation file: The *SmartSecTransformation.qvto* described in chapter 5.

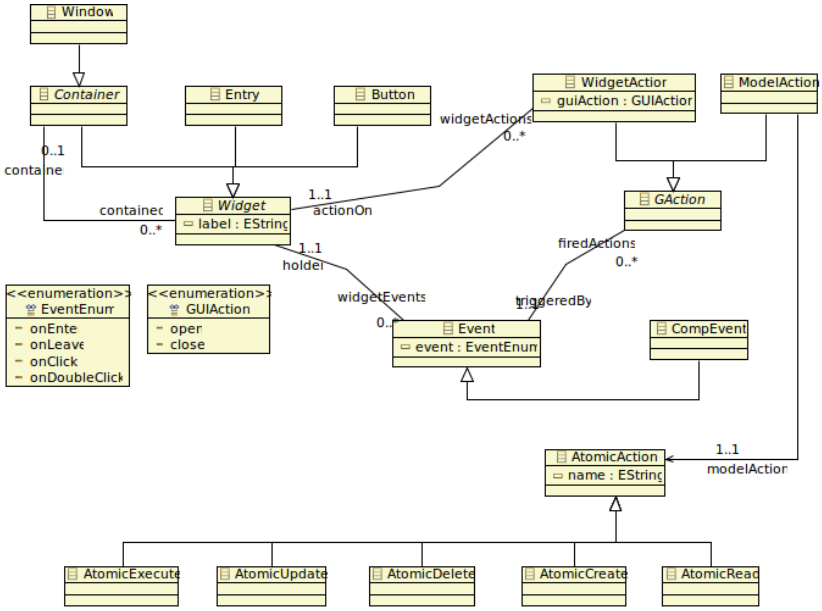

Figure D.1: GUI metamodel

D.3.2 Overview

The methodology using QVTO and eclipse is the following:

1. Creating empty EMF Projects
2. Importing the ecore Diagrams into the empty EMF Projects
3. Generating EMF Models in the EMF Projects
4. Generating the plug-in and the editor from the EMF Model
5. Running an Eclipse Application instance (preloaded with the generated plug-ins and editors)
6. Importing the input models
7. Importing the transformation files
8. Configuring the run configurations
9. Run the transformations

D.3.3 Metamodel Import

As a first step create three empty EMF Projects for the three ecore metamodels (GUI, SecureUMLandComponentUML, SecureUMLandGUI). We import now the three ecore metamodels from the file system into these projects. Right-click the folder *model* in one of the newly created projects. Choose *Import...*. In the window that opens choose

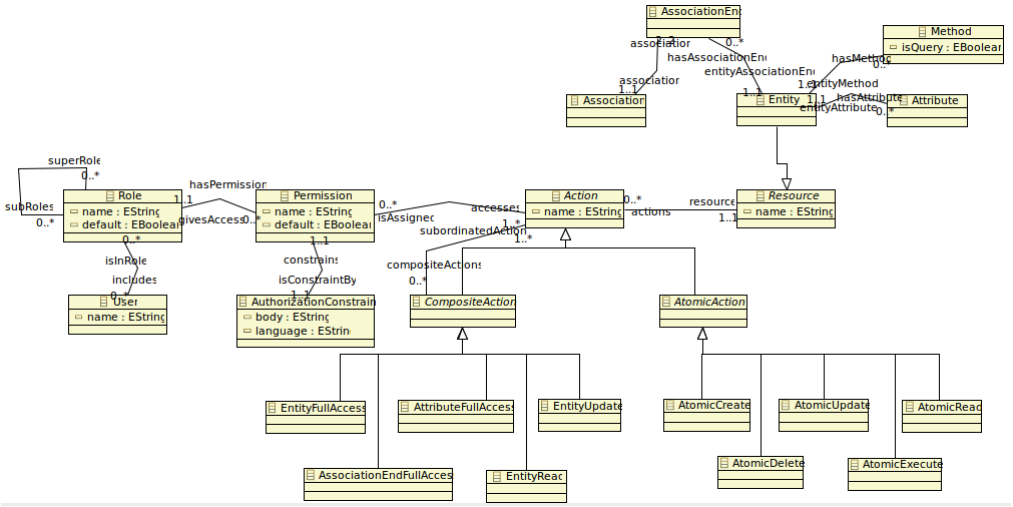

Figure D.2: SecureUML+ComponentUML Metamodel

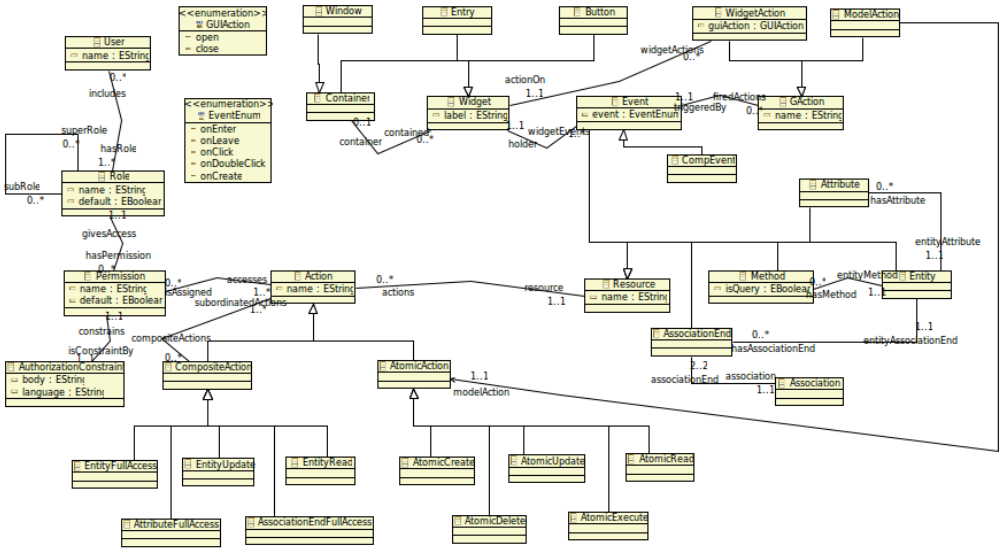

Figure D.3: SecureUML+GUI metamodel

General→File System and click *Next*. Now browse to the folder where the provided files are stored in your file system. Select in the wizard now the corresponding .ecore file and click *Finish*. The .ecore file is now available in the *model* folder. Repeat this step for all the ecore metamodels.

D.3.4 Creation of the EMF Models

Create a new EMF Model in each project. Therefore right-click the corresponding .ecore file and choose *New→Other...→EMF Model* and click *Next*. This file has to be named using the suffix *.genmodel* by default the name of the .ecore file is being used, if not type this name in by hand[2]. After clicking *Next*, choose *Ecore Model*, then click *Next*. The correct path to the .ecore model should already be proposed otherwise browse the workspace and choose the corresponding .ecore file, you have imported. Now choose *Load* and press again *Next* and in the next window the button *Finish*. A new EMF Model based on the ecore metamodel you imported has been generated. Repeat this step for the remaining ecore metamodels in the projects.

D.3.5 Plug-in and Editor Creation

Open one of the generated EMF Models. Right-click on the root entry and choose *Generate All*. This generates all the things needed in order to go on. A plug-in and an editor is being created, that can be used further on. Repeat this step for every EMF Model.

You can create several metamodels and plug-ins like described above. We recommend to create for every new metamodel a new empty EMF Project!

Now, as the plug-ins and editors are all created, choose *Run as...→Eclipse Application* (maybe you have to define the run configurations for running an eclipse application first). This runs a second instance of eclipse, automatically preloaded with all the defined plug-ins and editors.

D.3.6 Creation of Input models

As a first step you need to define a new Project in the second eclipse instance. Choose *Model to Model Transformation → Operational QVT Project* to create a new QVTO Project. This creates you a new Project with a folder transforms in it by default. There you can put .qvto files in to execute. We suggest you create a new folder called *models*.

Import the files *GuiModel.gui* and *SecurityPolicy.secureumlandcomponentuml* into the *models* folder. These are the input models of our transformations. You can view and edit these models by using the properties view maybe you have to show it using right-click.

Alternatively you can define your own input model(s). Choose *new→other ...→Example EMF Model Creation Wizard→yourmetamodelname Model*. Specify a name and the location to be stored. Then specify the root object of the model, i.e. *Model* and store it. A new file of type .yourmetamodelname has been created. With right-click on the root object you can include more elements into the model. By using the properties view at the bottom, the inserted objects can be initialized.

[2]If you want to use the provided example models, you have to use the names of the ecore models!

D.4 Transformations

D.4.1 Import of the transformation files

Now, that we have the metamodels and the input models as instances. We can take a look at the transformation files. Import the files *AggregationTransformation.qvto*, *SecurityTransformation.qvto* and *SmartSecTransformation.qvto* into the *transforms* folder.

D.4.2 Run Configurations

We are ready for the first transformation. Therefore, we have to specify, which models that have to be transformed to which output models. Right-click on the *AggregationTransformation.qvto* file. Choose *Run As→Run Configurations...* in the window choose *Operational QVT Interpreter* and create a new configuration. In this new configuration you can specify the input models, i.e. *GuiModel.gui* and *SecurityPolicy.secureumlandcomponentuml* by simply browsing to them. Now specify the output model, choose therefore the path to the *models* folder and call the output file for example *SecAwareGUI.secureumland-*
gui. Click apply and then run the AggregationTransformation. The resulting *SecAwareGUI.secureumlandgui* will be generated in the *models* folder. Double-click this file to analyze.

Now proceed as before for the SecurityTransformation and for the SmartSecTransformation. Choose there the file *SecAwareGUI.secureumlandgui* as the input/output model file in the run configurations.

Bibliography

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

[2] D. Basin, M. Clavel, M. Egea, and M. Schläpfer. Automatic generation of smart, security-aware GUI models. Submitted for publication, 2009.

[3] 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.

[4] K. Blankenhorn and W. Walter. Extending UML to GUI modeling. `http://www.bitfolge.de/pubs/MC2004_Poster_Blankenhorn.pdf`, 2004.

[5] M. Clavel, M. Egea, and M. G. de Dios. Building an efficient component for OCL evaluation. In *8th OCL Workshop at the UML/MoDELS Conference: OCL Concepts and Tools: From Implementation to Evaluation and Comparison*, ECE-ASST, Tolouse, September 2008.

[6] 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 '08: Proceedings of Model Driven Architecture - Industrial Track*, volume 5095 of *LNCS*, pages 327–338, Berlin–Heidelberg, 2008. Springer-Verlag.

[7] P. P. 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/`.

[8] R. Dvorak. *Model Transformation with Operational QVT*. Borland Software Corporation, 2008. `http://www.eclipse.org/m2m/qvto/doc/M2M-QVTO.pdf`.

[9] 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.

[10] M. P. Gallaher, A. C. OConnor, and B. Kropp. The economic impact of role-based access control. Technical report, National Institute of Standards and Technology-Acquisition and Assistance Division, 2005. `http://www.sis.pitt.edu/~jjoshi/TELCOM2813/Spring2005/RBAC_Economic_Impact.pdf`.

[11] 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.

[12] M. Hildebrand and J. van Ossenbruggen. Configuring semantic web interfaces by data mapping. In S. Handschuh, T. Heath, and V. Thai, editors, *Visual Interfaces to the Social and the Semantic Web (VISSW 2009)*, volume 443, February 2009.

[13] 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.

[14] A. Kleppe, W. Bast, J. B. Warmer, and A. Watson. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley, 2003.

[15] 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`.

[16] Object Management Group. Object Constraint Language specification. Technical report, OMG, May 2006. OMG document available at `http://www.omg.org`.

[17] M. Ogura, H. Mineno, N. Ishikaw, T. Osano, and T. Mizuno. Automatic gui generation for meta-data based pucc sensor gateway. In *Knowledge-Based Intelligent Information and Engineering Systems*, volume 5179 of *LNCS*, pages 159–166. Springer Berlin–Heidelberg, 2008.

[18] J. 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.

[19] 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.

[20] M. Schläpfer, M. Egea, D. Basin, and M. Clavel. Automatic generation of security-aware GUI models. In *ECMDA'09 Workshop: Security in Model Driven Architecture (SEC-MDA'09)*, June 2009.

[21] 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.

[22] 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 Second International Conference on Availability, Reliability and Security*, pages 644–654, Washington, DC, USA, 2007. IEEE Computer Society.

[23] 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.

[24] The Eclipse M2M Development Team. M2m-project. `http://www.eclipse.org/m2m/`.

[25] The Eclipse Platform. Eclipse download. `http://www.eclipse.org/downloads/`.

[26] W3C consortium. Resource description framework (RDF). `http://www.w3.org/RDF/`, 2004.

[27] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2st edition, 2003.